THE ARCHITECTURE AND DESIGN OF A HIGH LEVEL LANGUAGE PROCESSOR

by

ROGER BREES
B.S., Kansas State University, 1982

A MASTER'S THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1984

Approved by:

S E Hafft

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

Most computers being built are based on the concept presented by Von Neumann in the 1950's. This concept or architecture uses a single memory to store both data and the program instructions.

Typical machines today are also based on a register architecture. That is, there is some small number of high-speed general-purpose storage locations that can be used often for different functions. But the limited number means that many of the values in the registers must be shuffled back and forth from memory.

Programming languages, on the other hand, usually do not consider the program to be a part of the data. There are some exceptions to this. LISP, for example, is happy to rewrite the program thinking that it is modifying a list of some sort. The Block-structured languages (those derived from ALGOL especially) model a computer as a Harvard machine, i.e., a machine that has two independent memories - one for the data and one for the instructions. Thus the program can never modify the program.

Block-structured programming languages do not usually support the concept of a register. They define variables and constants when a block is entered, then use these for the storage of data. Further, the variables and the constants defined in the block disappear or are deactivated when the

block finishes. Some of the variables of other blocks can be referenced by the executing block. Others cannot. The scope of the variables is that part of the program in which they can be referenced. The rules defining when a variable can be referenced and when it cannot are clearly defined. By using a stack the scope of the variables can be maintained as the program executes.

The differences between the model of the computer seen by the programming language and the actual machine architecture is usually bridged by the compiler. The compiler is responsible for maintaining the variable scope in the register machine. The scope must be maintained as blocks are declared and dropped and as procedures are called and returned from. All this requires complex manipulations of pointers and lists.

This paper presents the design of a computer which closely supports block-structured languages. The machine has separate instruction and data memories, uses a stack to maintain the scope of variables during execution, and has an instruction set which resembles the instruction set of the block-structured languages.

II. OBJECTIVES

The primary objective of this paper is to present the architecture and a possible implementation of a computer that closely supports high-level, block-structured programming languages.

The three features that have been given precedence are;

1. Support for high-level constructs
2. Ease of use
3. Simplicity of design

Other features such as speed and hardware efficiency have been given secondary consideration.

The discussion and presentation of the design assume that the reader is familiar with digital design procedures. The features of the various processors used are not discussed, but are widely available in the literature (1),(2). The design is given in enough detail that it can be constructed.

No attempt has been made to incorporate some of the features that might be necessary to make this a "useful" machine. For example, characters are supported only in that the Input/Output can be made to treat the data it is reading or writing as a character. Characters can be handled without any special hardship, but they are inefficient in terms of the memory used.

III. ARCHITECTURE OVERVIEW

Most of the block-structured programming languages are implemented using a stack. The computer described in this paper supports the stack concept directly. Its stack can be modelled as three separate stacks, each with a distinct function, and it implements these stacks as the major part of its architecture.

The first and simplest of the three stacks is the operand stack. An operand stack is used to hold the operands and results of operations. An operation, such as ADD, that requires two operands, uses the top two elements of the stack. The result of the operation is put back onto the top of the stack. Figure 1 shows the stack during an ADD operation. The level of the stack is reduced by one because the two operands are removed then the result put on.

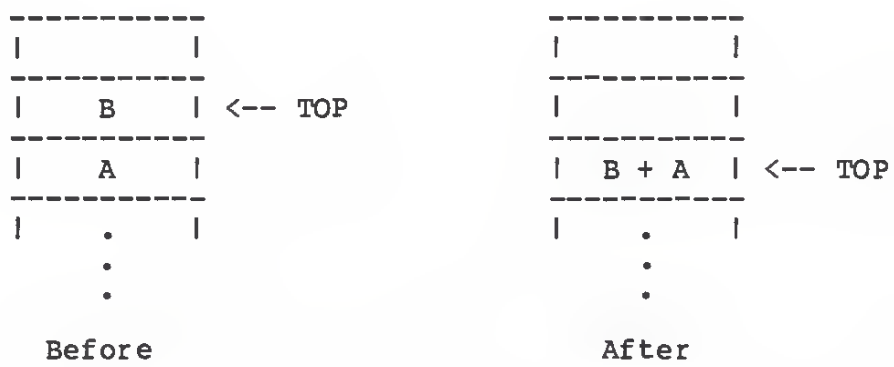
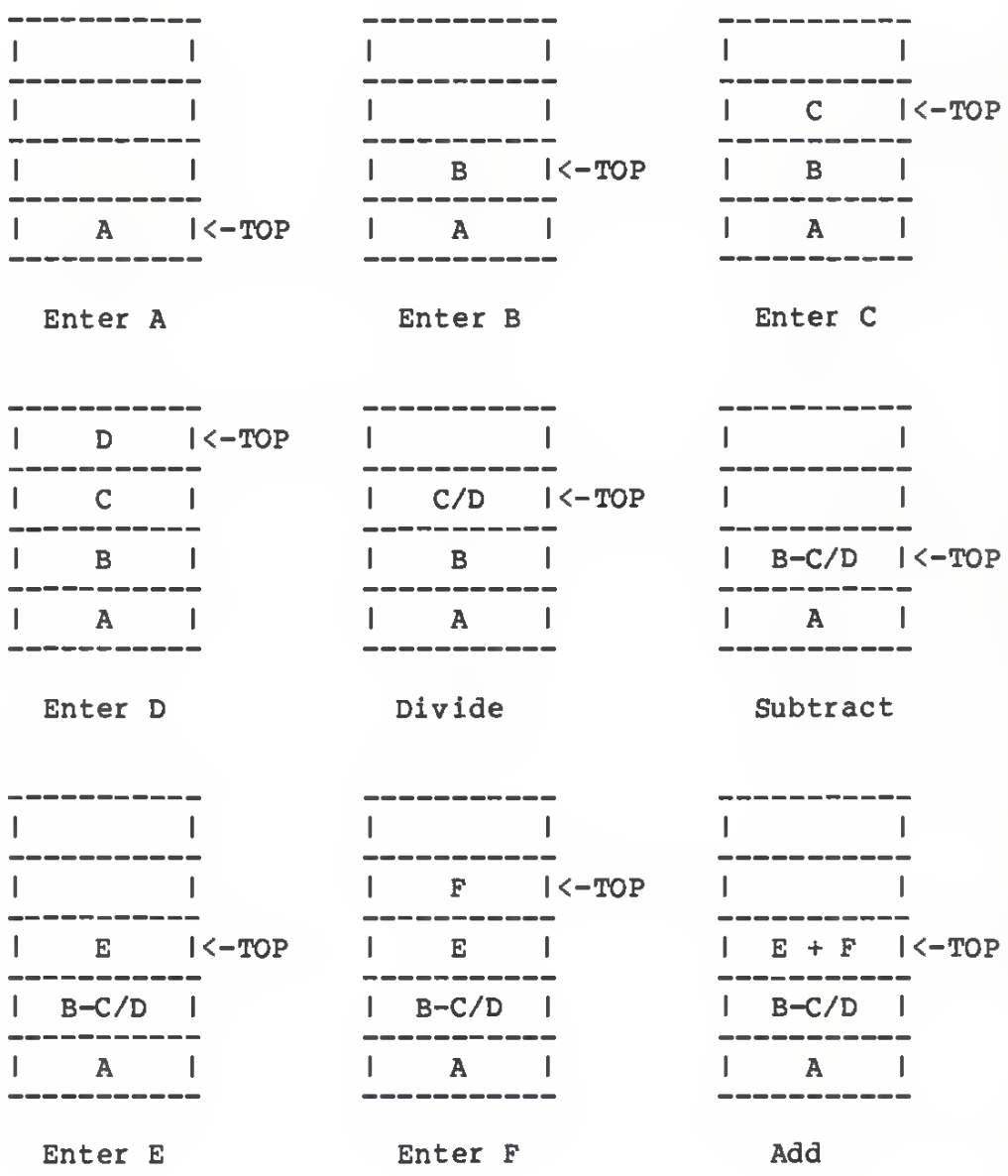


Figure 1. Operand Stack for an Add Operation

By using an operand stack, the instruction that performs an operation does not need to specify where the operands come

from, nor where the result goes. These are always the same place: the top of the stack. A more complex example of the operand stack is shown in Figure 2. This example uses an operand stack to evaluate the expression A+(B-C/D)*(E+F).



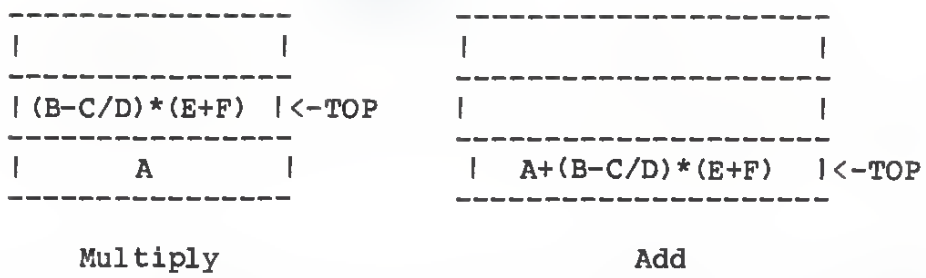
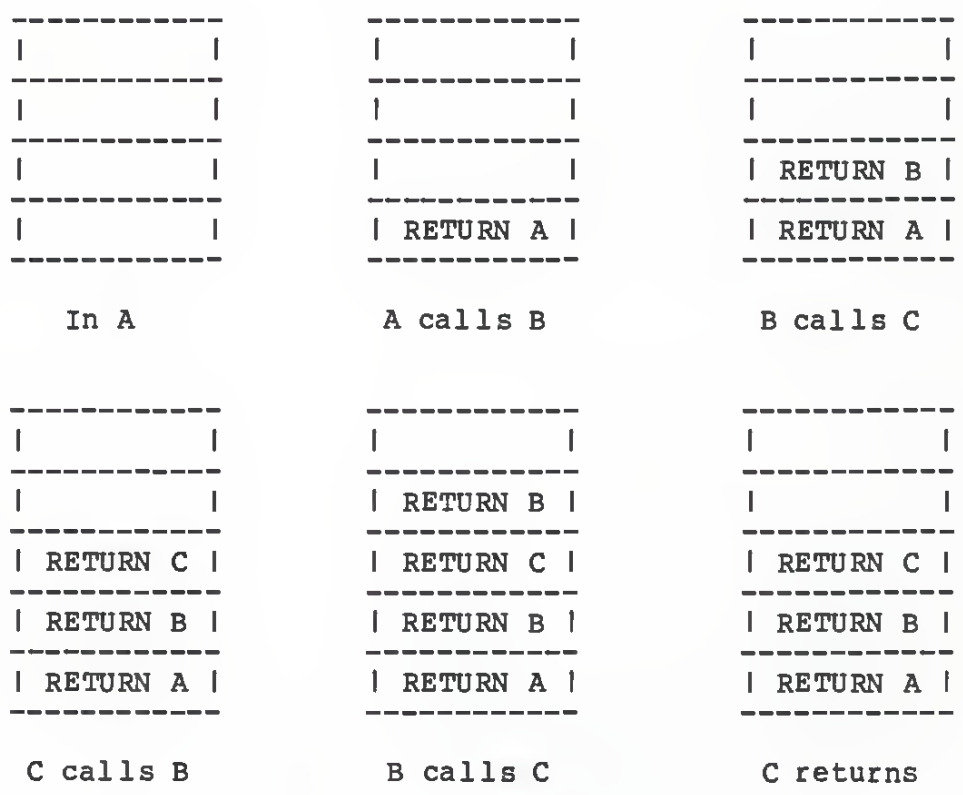


Figure 2. Operand Stack used in Evaluating an Expression

The second kind of stack used in high-level languages is a control stack. This stack is used to store linkage information when a procedure or subroutine is called. A procedure call causes a branch out of the instruction stream of the calling routine to that of the called routine. When the procedure has finished executing, it must cause a branch back into the calling routine. This return address could be stored in a dedicated memory location or in a register. However, if either of these were used the information would be lost if the called routine in turn called another routine. The return address could also be stored in a dedicated location local to the called routine. This breaks down if the routine ever recurses, (i.e. directly or indirectly calls itself) because the second return address is stored over top of the first. The other solution is to use a stack to hold the return addresses. When the subroutine is called, the return address is pushed onto the control stack. Then if more routines are called, their return address are also pushed onto the stack. When the routines finish and return to the routine that called them, the return addresses are popped off the stack.

Consider a main routine A which calls a subroutine B. B, somewhere in the course of its execution, calls another subroutine, C. Suppose that C calls subroutine B if some condition is met. If the condition is met on the first instance of C, then C calls B and B has recursed. Figure 3 shows the control stack for the case where the condition is met the first time but fails after that.



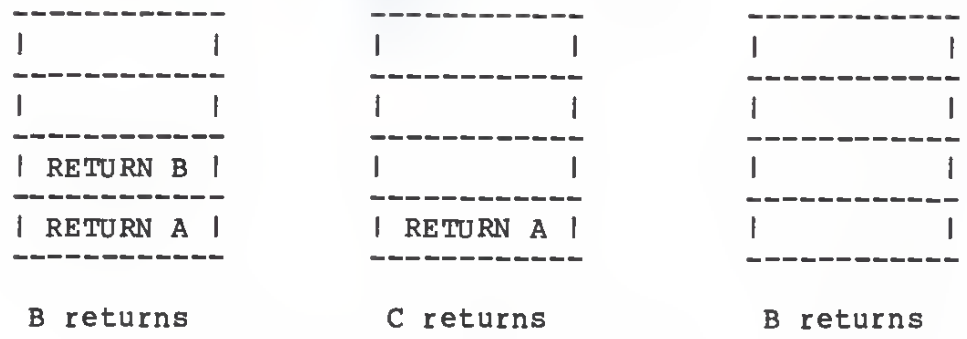


Figure 3. The Control Stack.

When a routine finshes executing it returns to the calling routine by removing the return address at the top of the stack and branching to that location. The correct order is maintained by the nature of the Last-In-First-Out stack. The correct return address is always the one at the top of the stack when the routine finishes execution, although; many subroutines may have been called in between.

The third and most complex stack used in high-level languages is a variable stack. A variable stack is used to hold variables. The complexity arises from the necessity to maintain the scope of the variables. The scope of a variable is simply that part of the program in which that variable is known. The usual definition of scope is that a variable is known to all blocks declared within the same block as the variable, unless a new variable of the same name is declared. In most ALGOL-derived languages a block may be in-line. That is, it may appear in the code as a section, set off by BEGIN and END statements, that may have variable names local to

itself. Example 1 on the following page demonstrates the scope of variables.

In the example, variables i, j, and m are declared in procedure A. Therefore they are known to all procedures declared within A unless those procedures declare a new variable with the same name. In the example, procedure B does declare a new variable named i. Thus the i declared in A is inaccessible in B. The same scope rules apply to the new instance of i. It is known to all routines declared within procedure B unless they declare a new variable i. Procedure C does not declare any variable named i, so the one in B is known in C. The variable k, declared in procedure B, is also known to both procedures B and C.

The procedure named D is outside of both procedures B and C. So the variables of B and C are unknown in D, and those of D are unknown in B and C. An application of the scope rules to the variables of D shows that i and m from procedure A and j and l from procedure D are known to procedure D. Procedure E redeclares the variable i, so it is a different version from the one in A and D. All this is shown in Table 1. All variables known to a procedure are given and the procedure in which that instance of the variable was declared is given in parentheses.

```
PROGRAM A;
     VAR i,j,m : INTEGER;

     PROCEDURE B;
          VAR i,k : REAL;

          PROCEDURE C;
               VAR j:REAL;
               BEGIN
                .
                .
                .
               END;        (*End of procedure C*)

          BEGIN
           .
           .
           .
          END;             (*End of procedure B*)

     PROCEDURE D;
          VAR j,l:REAL;

          PROCEDURE E;
               VAR i:INTEGER;
               BEGIN
                .
                .
                .
               END;        (*End of procedure E*)

          BEGIN
           .
           .
           .
          END;             (*End of procedure D*)

     BEGIN
      .
      .
      .
     END;                  (*End of main routine A*)
```

Example 1. A Program to Demonstrate Scope

These scope rules can be applied to more than just variable names. Procedure names, constants, functions, etc. all have scope within a program.

Procedure	Variables
A	i(A) j(A) m(A)
B	i(B) j(A) k(B) m(A)
C	i(B) j(C) k(B) m(A)
D	i(A) j(D) l(D) m(A)
E	i(E) j(D) l(D) m(A)

Table 1. The Variables Known to theProcedures

When a block is entered during execution, space is set aside on the variable stack for the variables local to the block. The location of the variables in the stack will depend on the run time behavior of the program, but the variables will always be a fixed distance from the first location reserved on the stack for the block. A pointer to the first location is saved. A block is able to access all variables declared outside of it if those variables are included in the scope of

the block. Therefore a pointer to each of the variable spaces of the outer blocks must also be kept. These pointers can be kept in registers called display registers.

The pointers that must be available are dependent on the static hierarchical structure of the block declarations and not on the run time behavior. The hierarchy of declarations for Example 1 is shown in Figure 4. Each level of declarations is called a lexical level. A block has access to the variables of only one block at each lexical level that is lower than or equal to its own. A block may not access any variables in a block whose lexical level is higher than its own. Thus there is one display register for each lexical level. The lower level display registers need to be set to the blocks with lower lexical levels than the curent block. Table 2 shows how the display registers are set for each procedure in Example 1. LL(1x) indicates the display register for lexical level 1x. LLTOP indicates the current lexical level, and points to the highest valid display register. Figure 5 shows how these pointers all work together to define parts of the variable stack for each block.

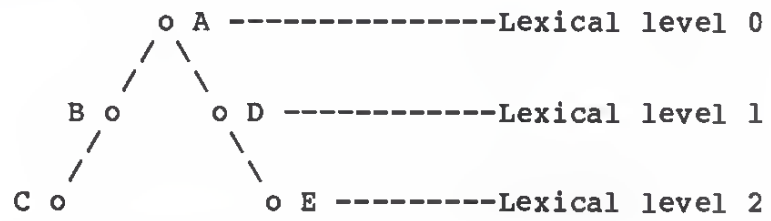


Figure 4. The Lexical Levels of the Procedures.

Procedure	Display Registers
A	LL(0) to A
B	LL(0) to A LL(1) to B
C	LL(0) to A LL(1) to B LL(2) to C
D	LL(0) to A LL(1) to D
E	LL(0) to A LL(1) to D LL(2) to E

Table 2. The Display Register Settings

Once the display is set for a given block, the variables can be accessed by the double lx,n, where lx selects the display register corresponding to lexical level lx, and n is the index into that variable space to the desired variable. The location of a variable in the stack may not be known at run time, but it can always be referenced by the two-part address lx,n. This address is known at compile time. Table 3 repeats Table 2 with the addition of the address double for each variable.

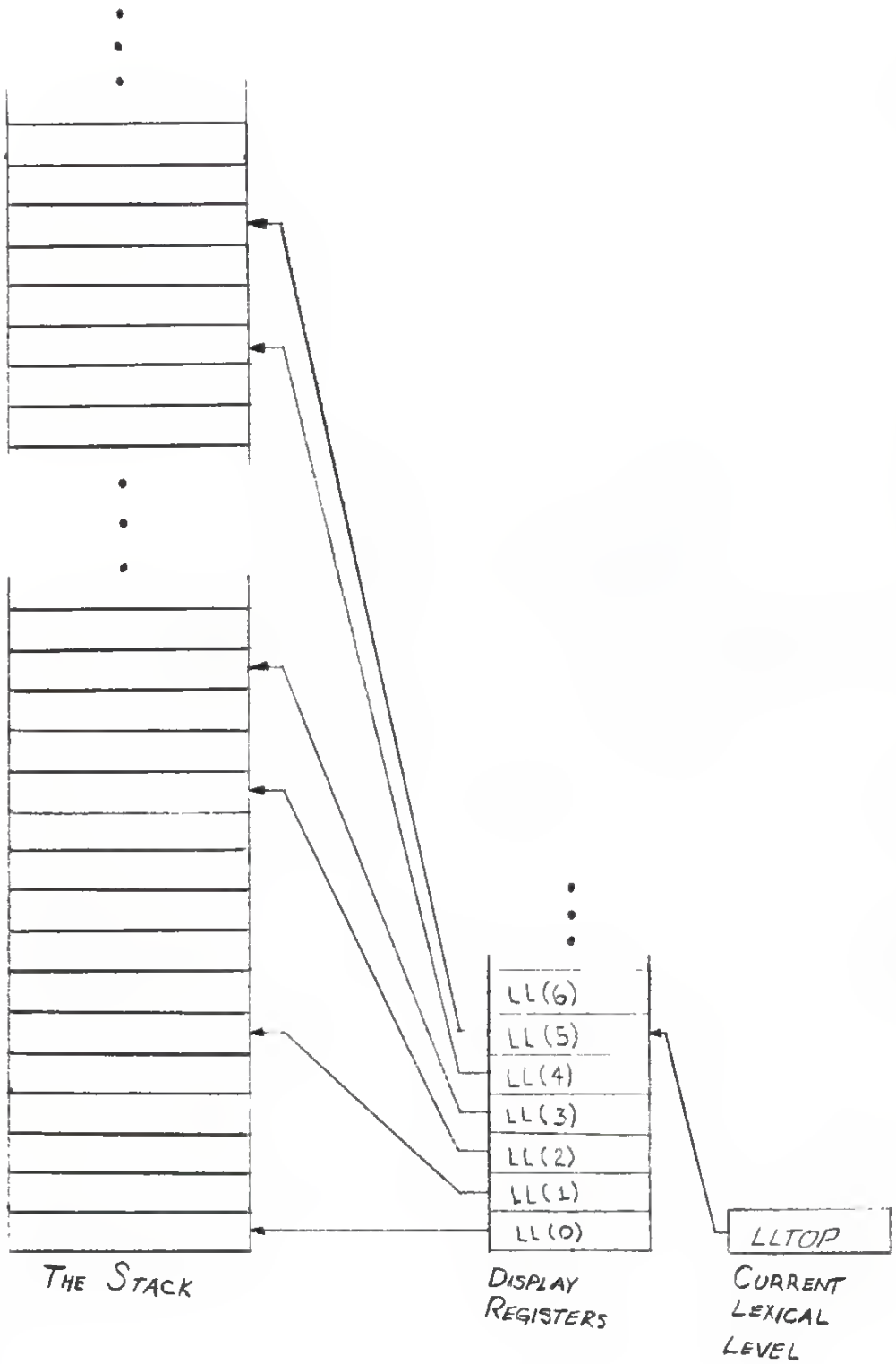


FIGURE 5. THE DISPLAY

Procedure	Variables	Address
A	i(A)	LL(0),0
	j(A)	LL(0),1
	m(A)	LL(0),2
B	i(B)	LL(1),0
	j(A)	LL(0),1
	k(B)	LL(1),1
	m(A)	LL(0),2
C	i(B)	LL(1),0
	j(C)	LL(2),0
	k(B)	LL(1),1
	m(A)	LL(0),2
D	i(A)	LL(0),0
	j(D)	LL(1),0
	l(D)	LL(1),1
	m(A)	LL(0),2
E	i(E)	LL(2),0
	j(D)	LL(1),0
	l(D)	LL(1),1
	m(A)	LL(0),2

Table 3. The Variable Addresses

The scope rules can be applied to procedure names as well as variable names. Thus for Example 1, procedure E could have called procedure B. Procedure E is at lexical level 2, while procedure B is at lexical level 1. The display registers which define the scope of procedure E must be changed during such an "up-level" call to set the proper scope for procedure B. The display registers before the call to B, are shown in Figure 6A. The display registers, as they should be set after the call to procedure B are shown in Figure 6B.

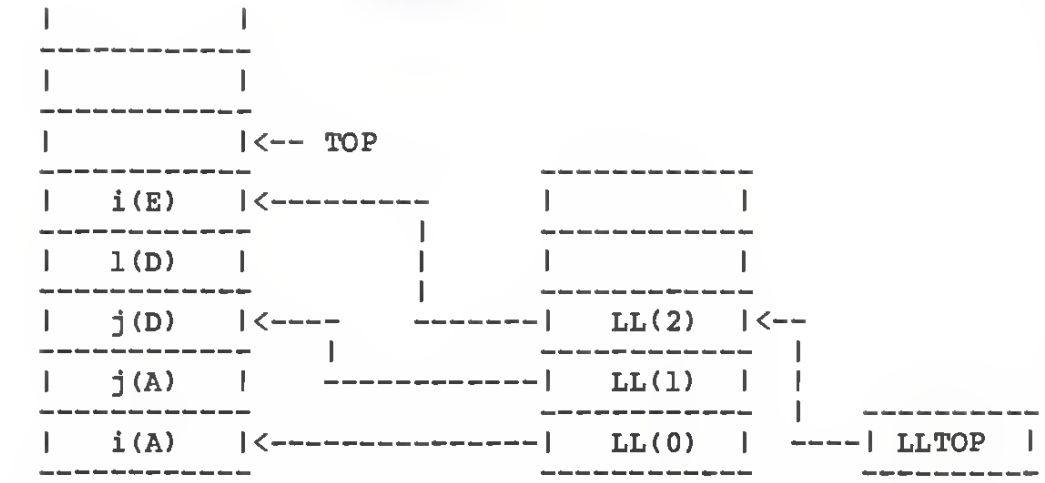


Figure 6A. The Display Before the Call to Procedure B

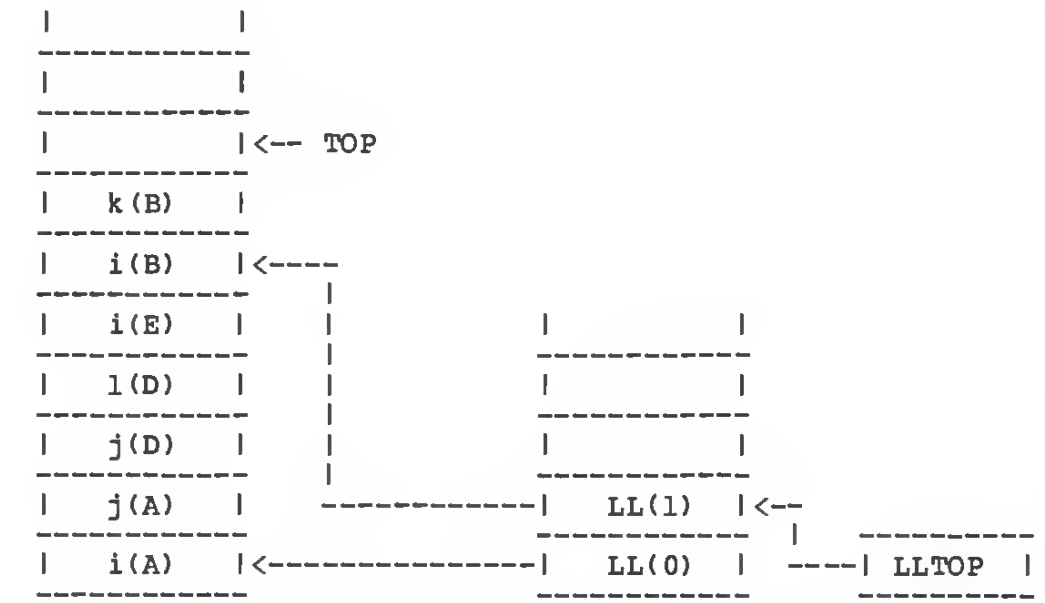


Figure 6B. The Display After the Call to Procedure B

The call mechanism needs to set the display pointers so the scope of the destination is correct. It must also save enough information to allow the scope of the calling routine to be restored once the called procedure finishes.

It is simple to set the display registers so that the scope of the destination is correct. The call needs to look as if it came from the block in which the called procedure was declared. This will ensure that the scope is correct. The current lexical level is set to be that of the outside block. The called procedure performs a block entry which increments the current lexical level, then sets the display register for that level. The lexical level of the block in which a procedure is declared is known at compile time and can be included as part of the procedure entry instruction.

The values that must be saved in order to restore the scope of the calling routine are the lexical level, and the value of the display register for the calling routine. However, these are insufficient to completely restore the scope if the called routine is one or more levels higher than the calling block. For example, if a procedure at level 4 calls a procedure at level 2, all display registers from 2 to 4 must be restored to ensure that the scope of the calling routine is correct after the called procedure has finished. This requires that the display register be copied into the stack before a new block is entered. For the procedure call from level 4 to level 2 the variable stack with the linkages might be as shown in Figure 7.

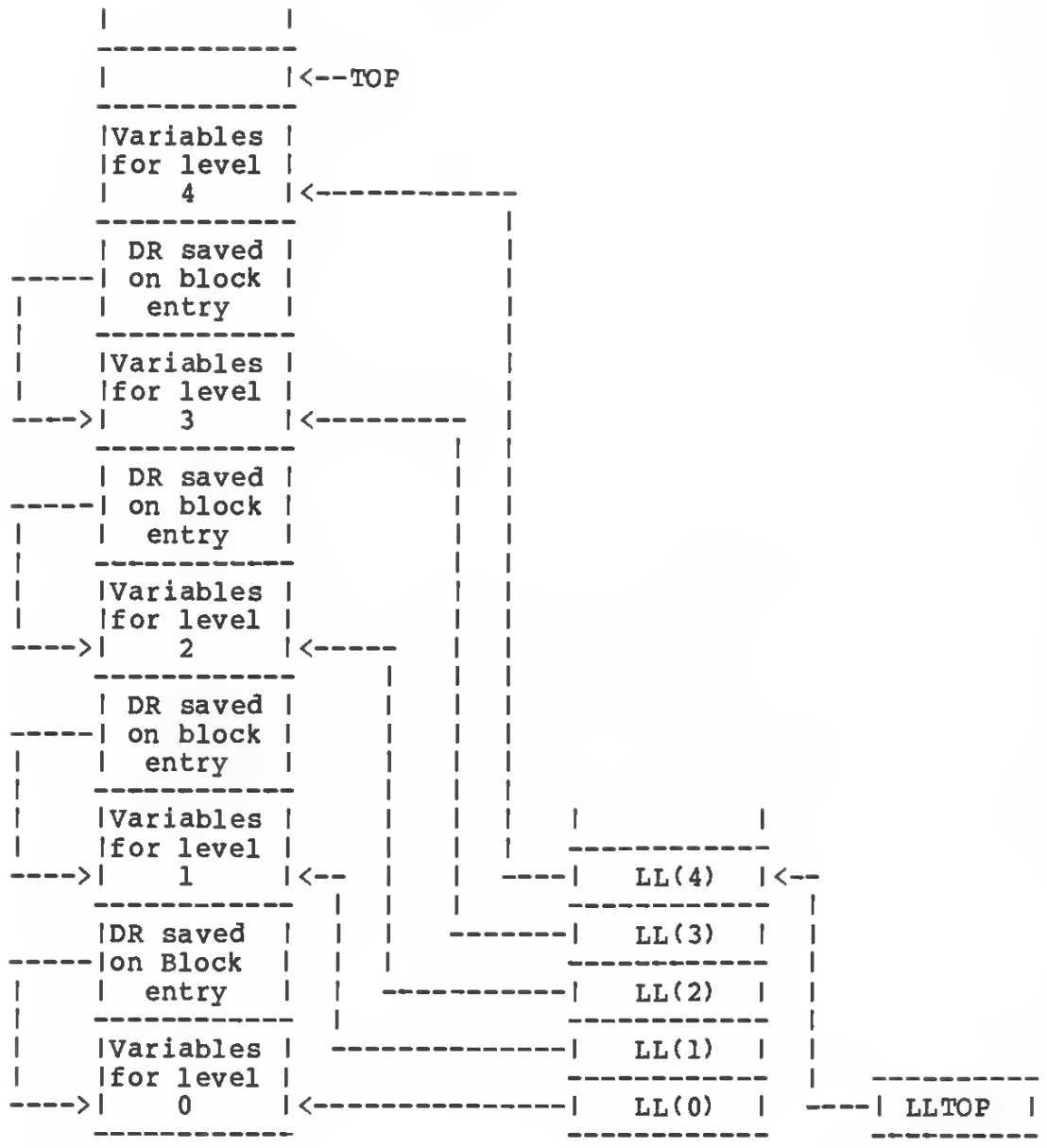


Figure 7A. The Variable Stack Linkages Before the Call

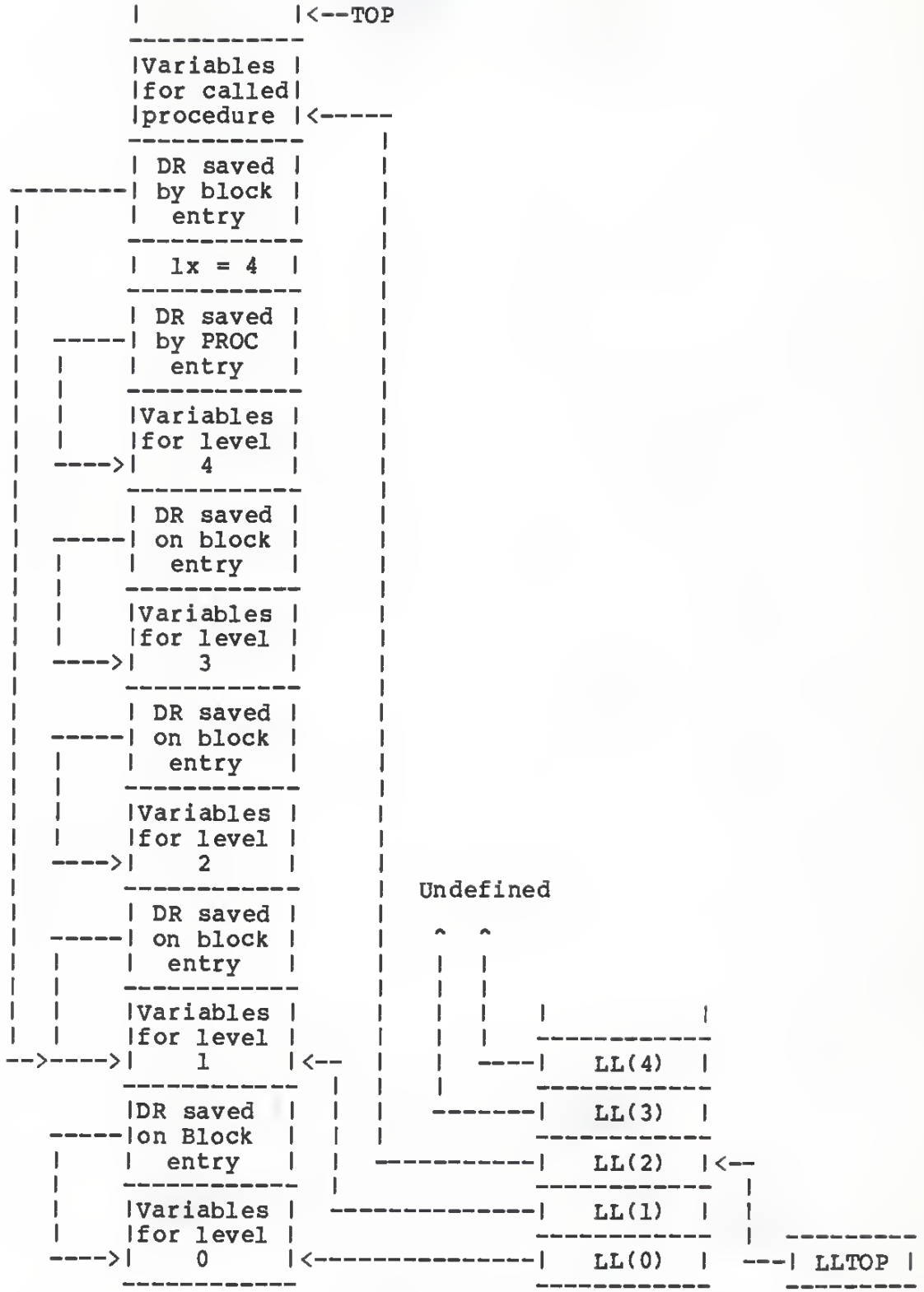


Figure 7B. The Variable Stack Linkages Just Before the Return

To completely restore the scope of the calling routine, the lexical level and the display register for the calling block are restored from the stack. Then the next lower display register is restored by loading it with the value just below where the newly restored display register points. The next lower level pointer is restored in the same way. The relationship between the just-restored display register and the next lower display register can be written as

```
LL(lx-1) <-- S(LL(lx)-1)
```

The entire sequence for the restoration of the scope of the calling routine is;

```
LL(4) <-- S(TOP)  Popped off the stack by Procedure Exit
LL(3) <-- S(LL(4)-1)
LL(2) <-- S(LL(3)-1)
```

This can be written as a procedure.

```
i = lx;
WHILE ( i>0 ) DO;
    LL(i-1) = S(LL(i)-1);
    i = i-1;
END WHILE;
```

The three kinds of stacks can be combined into a single physical stack. The display linkage information and the return address are stored together during a procedure call. This triple of values is called a Transfer Point. The operand stack sits on top of the variable stack. For the combined stack, a procedure entry requires the following steps:

1. Save the current display register.
2. Save the current lexical level.
3. Save the return address.
4. Set the lexical level to that in which the procedure was declared.
5. Do a block entry.

The steps necessary when entering a block are:

1. Save the current display register.
2. Increment the lexical level.
3. Set the new display register.
4. Reserve the variable storage on the stack.

The class of languages derived from ALGOL allows a procedure or block to declare variables which retain their values from one invocation to the next. Pascal does not have this feature. To implement a random number generator in Pascal, the seed has to be passed back and forth to the procedure as a parameter, because it can not be retained in the procedure from one call to the next. In PL/I, the seed can be declared as an "OWN" variable. It can then be set by a call to the random number generator, and it will retain that value on the next call to the procedure. The calling program need not carry the variable SEED, which means nothing to it.

OWN variables can not be implemented with a stack. A HEAP is used to store variables which must not disappear between invocations of the procedure in which they are declared. The HEAP is also used for PL/I's "BASED" variables, which are normally used to implement list and tree structures.

The HEAP tends to become cluttered with areas which were in use at one time, but have since been discarded. PL/I uses

the functions ALLOCATE and FREE to claim and release blocks of memory. For operations of any extent which use the heap, garbage collection is necessary. There are many algorithms available for garbage collection (3).

IV. INSTRUCTION SET

There are two data types in this implementation. The numeric data type is used to represent integer numbers. (There are no floating point numbers in this implementation.) The other data type is logic. A value of the logic data type can have only two values; TRUE and FALSE.

The instruction set for this computer is aimed at the ALGOL-like languages. It allows complete implementation of the ALGOL instruction set. The instruction set is presented in "Compiler Construction: Theory and Practice" by Barrett and Couch (4). Pascal is mainly a subset of ALGOL, so it, too, should be completely supported. The instructions can be divided into 8 catagories. These are:

1. Arithmetic instructions
2. Comparison instructions
3. Logic instructions
4. Data movement instructions
5. Stack and Heap maintenance instructions
6. Block maintenance instructions
7. Jump instructions
8. Input/Output instructions

An arithmetic instruction is one which causes some arithmetic operation to be performed. Any operand is always taken from the top of the stack and the result is always put onto the top of the stack. The operands and result must be of the numeric data type. The arithmetic instructions are:

ADD - Pop the top two stack elements, add them, and put the result onto the stack.

SUBTRACT - Pop the top two stack elements, subtract thetop stack elementfrom the second element, and put the result ontothe stack.

MULTIPLY - Pop the top two stack elements, multiply them together, and put the product onto the stack.

DIVIDE - Pop the top two stack elements, divide the second by the first, and put the quotient and remainder back onto the stack.

NEGATE - Pop the top stack element. Negate it, and push the result back onto the stack.

The comparison instructions pop two numeric operands from the stack and push a logic result whose value depends on the result of the compare. The comparsion instructions are:

EQUAL - Pop the top two numeric operands. Compare them. Push a TRUE result if and only if both operands are the same. Otherwise push a FALSE result.

GREATER THAN - Pop the top two numeric operands. Compare them. Push a TRUE result if and only if the second operand is larger than the top operand. Otherwise push a FALSE result.

LESS THAN - Pop the top two numeric operands. Compare them. Push a TRUE result if and only if the second operand is less than the top operand. Otherwise push a FALSE result.

The logic instructions require operands of the logic data type and produce a result of the logic data type. The operands are popped off the stack and the result is pushed onto it. The logical instructions are:

NOT - Pop the top operand. Complement it and push the result back onto the stack.

AND - Pop the top two operands off the stack. Logically AND them. Push the result back onto the stack.

OR - Pop the top two operands off the stack. Logically OR them. Push the result onto the stack.

The data movement instructions move data onto or off of the stack or heap. Some instructions require an operand. The operand can be of either data type or it can be an address. The data movement instructions are:

LOAD - Pop an address off the top of the stack. Fetch the value stored in that address and push that value onto the stack.

LOAD CONSTANT c
- Push the constant c onto the stack.

LOAD ADDRESS lx,j
- Push the absolute address of the variable specified by the pair lx,j onto the stack.

LOAD LABEL z
- Load a transfer point onto the stack. (More on labels later.)

LOAD DATA lx,j
- Push the value found in the variable whose address is lx,j onto the stack.

LOAD HEAP j
- Push the value found in location j in the heap onto the stack.

LOAD HEAP ADDRESS j
- Push the address of element j of the heap onto the top of the stack.

REPLACE
- Pop the top of the stack. Pop an address from the new top of the stack and push the value pointed to by that address onto the stack. This instruction is only used by the Call-by-name parameter mechanism.

STORE - Pop the top element off the stack. Store it into the location whose address is at the new top of the stack. Pop the address off the stack.

STORE DATA lx,j
- Pop the top element off the stack and store that value into the variable whose address is lx,j.

STORE HEAP j
- Pop the top element off the stack. Store that value into location j in the heap.

The Stack and Heap maintenance instructions change the value of the stack and heap pointers. They are:

INCREMENT AND SAVE STACK
- Pop the top element off the stack. Add that value to the top of stack pointer to form a new top of stack. Save the original value of the top of stack at the new top of the stack.

INCREMENT STACK n
- Add n to the top of the stack pointer.

DECREMENT FROM STACK
- Pop the top element off the stack. Subtract that value from the top of stack pointer.

DECREMENT STACK n
- Subtract n from the top of stack pointer.

INCREMENT AND SAVE HEAP
- Pop the top element off the stack. Add that value to the bottom of the heap pointer. Push the new heap pointer value onto the top of the stack.

INCREMENT HEAP n
- Add n to the Heap pointer.

DECREMENT AND SAVE HEAP
- Pop the top element off the stack. Subtract that value from the bottom of the heap pointer. Push the result onto the stack.

DECREMENT HEAP n
- Subtract n from the Heap Pointer.

The Block instructions handle the display chain and variable spaces. They are:

BLOCK ENTRY n
- Save the display register, increment the lexical level, set the new display register, and reserve space on the stack for the variables of the block. (The entire sequence for a block entry is discussed below in the section on labels.)

BLOCK EXIT
- Remove all the information saved on the block enter. (See below.)

PROCEDURE ENTRY lx
- Enter procedure at lexical level lx. (See below)

LABEL ENTRY lx
- Set lexical level and stack pointer for the label. (See below)

The jump instructions cause a branch in the program flow. They are:

JUMP z
- Jump to location z in the instruction stream.

CONDITIONAL JUMP z
- Pop the top stack element. Jump to location z if the value is FALSE, otherwise fall through to the next instruction.

JUMP INDIRECT lx,j
- Jump to the destination whose address is stored in a variable whose address is lx,j.

CALL PROCEDURE z
- Jump to the procedure at z and save the return and display information. (see below)

CALL PROCEDURE INDIRECT lx,j
- Jump to a procedure whose address is in a variable. Save the return and display information.

RETURN
- Branch to the place from which a procedure was called and restore all the display information. (See chapter 3)

The Input and Output instructions cause a single value to be read or written. All I/O instructions require a device code. The device code can be at the top of the stack or in the instruction word immediately following the I/O instruction. The Input/Output instructions are:

READ Q
- Read a numeric value from device Q and put it on the stack.

WRITE Q
- Pop the top element off the stack and write its value to device Q.

READ CHARACTER Q
- Read a character from device Q and push it onto the stack.

WRITE CHARACTER Q
- Pop the top element off the stack and write it out to device Q as a character.

READ STACK
- Pop a device code off the stack, then read a numeric value from that device and push it onto the stack.

WRITE STACK
- Pop a numeric value off the stack, then write it out to a device whose device code is also popped off the stack.

READ CHARACTER STACK
- Pop a device code off the stack and read a character from the selected device. Push the character onto the stack.

WRITE CHARCTER STACK
- Pop a character off the stack. Write it out to a device whose device code is popped off the stack.

The LOAD LABEL instruction saves the current lexical level pointer, the current lexical level, and an instruction address on the stack. The three values saved by the LOAD LABEL are the same three values that are saved by the Procedure mechanism. This triple is called a transfer point. The label is placed on the stack so that the CALL PROCEDURE INDIRECT instruction can have a branch target. This indirect call is used during parameter evaluation.

Other jump instructions cause a change in program flow but no return to the calling location is required. A transfer point does not need to be loaded onto the stack and any information in the stack above the variable space for the destination block is no longer needed. The jump mechanism loads the top of the stack pointer with the location just above the last variable. This is called the working pointer. The working pointer (WP) is saved by the Block Entry sequence. The Block Entry is then modified from that given in Chapter Three. The final form of the Block Entry sequence for a block having n variables is :

1. Save the value of the top of stack pointer after the block entry is completed. (WP = TOP+n+2)
2. Save the current lexical level pointer.
3. Increment the current lexical level.
4. Set the current lexical level pointer to point to the current top of the stack.
5. Set the top of stack pointer to point to the first location above the variables. (TOP+n)

Everytime something is pushed onto the top of the stack, the top of stack pointer is incremented. So the value saved in step 1 and the value loaded into the Top of stack pointer in step 5 are the same value. Two values have been pushed onto the stack in between.

The order is important in many operations. If a jump out of a block is desired, the jump must be preceded by a LABEL ENTRY instruction to set the lexical level and the stack pointer to what they should be for the destination. A LABEL ENTRY performs the following operations:

1. Set the current lexical level to that of the destination.
2. Set the top of stack pointer to the value saved by the block entry of the destination block.

Any call to a procedure must include a PROCEDURE ENTRY instruction to set the lexical level to look as if the call came from the block in which the procedure was declared. This ensures that the scope is correctly set. The order for a procedure call then is:

1. CALL PROCEDURE (in the calling block)
2. PROCEDURE ENTRY (in the destination procedure)
3. BLOCK ENTRY (in the destination procedure)

The JUMP INDIRECT instruction is included to support an archaic ALGOL construct. A label can be passed to a procedure as a parameter. The procedure can then do a jump to that label. The return is bypassed and there is no guarantee that the program flow will ever reach the instruction after the call. This violates the philosophy of block-structured languages. The JUMP INDIRECT instruction must set the scope for the destination in the same manner that the procedure RETURN sets the scope.

Barrett and Couch give numerous examples of these instructions and their use in implementing the constructs of the ALGOL language.

The instruction set has forty-eight instructions. The instructions, their mnemonics, and their op-codes are given in Table 4. An instruction is one or two instruction words long. The first half of the first word is always an op-code. This is

all that is necessary for some instructions. If nothing more is required, the last half is ignored. If needed, the second half is the lexical level. This is used by all the instructions that reference a variable location and by the LABEL ENTRY and PROCEDURE ENTRY instructions. The second word of an instruction, when used, has several meanings depending on the op-code in the first word.

The instructions are given in the format:

INSTRUCTION (operand),(operand)

The operands are those that are given in the instruction stream. All operands require one instruction word except the lexical level (lx), which is in the second half of the instruction word containing the op-code.

The abbreviations used in Table 4 are:

lx - lexical level
j - variable index
c - a data constant
z - a jump destination address
n - a pointer displacement
Q - an Input/Output device code

The op-codes are determined in part by the requirements of the mapping logic. (See chapter IX.)

INSTRUCTION	MNEMONIC	OP-CODE
ADD	ADD	40
SUBTRACT	SUB	41
MULTIPLY	MULT	42
DIVIDE	DIV	43
NEGATE	NEG	20
EQUAL	EQU	44
GREATER THAN	GREAT	45
LESS THAN	LESS	46
NOT	NOT	21
AND	AND	47
OR	OR	48
LOAD	LOAD	22
LOAD CONSTANT c	LCONST	80
LOAD ADDRESS lx,j	LADD	81
LOAD LABEL z	LLAB	F0
LOAD DATA lx,j	LDATA	82
LOAD HEAP j	LHEAP	83
LOAD HEAP ADDRESS j	LHADD	84
REPLACE	REP	49
STORE	STOR	4A
STORE DATA lx,j	SDATA	24
STORE HEAP j	SHEAP	25
INCREMENT-SAVE STACK	INCS	F1
INCREMENT STACK n	ISTAK	F2
DECREMENT FROM STACK	DECS	F3
DECREMENT STACK n	DSTAK	F4
INCREMENT-SAVE HEAP	INCH	26
INCREMENT HEAP n	IHEAP	02
DECREMENT-SAVE HEAP	DECH	27
DECREMENT HEAP n	DHEAP	03
BLOCK ENTRY n	BLKEN	F5
BLOCK EXIT	BLKEX	F6
PROCEDURE ENTRY lx	PROCEN	F7
LABEL ENTRY lx	LABEN	F8
JUMP z	JUMP	01
CONDITIONAL JUMP z	COND	23
JUMP INDIRECT lx,j	JIND	F9
CALL PROCEDURE z	CALL	FA
CALL PROCEDURE INDIRECT z	CALIN	FB
RETURN	RET	FC
READ Q	READ	85
WRITE Q	WRITE	28
READ CHARACTER Q	READC	86
WRITE CHARACTER Q	WRITEC	29

INSTRUCTION	MNEMONIC	OP-CODE
READ STACK	RSTAK	2A
WRITE STACK	WSTAK	4B
READ CHARACTER STACK	RSTAKC	2B
WRITE CHARACTER STACK	WSTAKC	4C

Table 4. The Instruction Set

V. IMPLEMENTATION OVERVIEW

The hardware design of this machine tries to parallel the architectural model. There are separate memories for the instructions and the data. Both the stack and heap reside in the data memory. The stack grows up from the bottom of memory and the heap grows down from the top of memory. The full display, as described in chapter three, is implemented. The display registers occupy the bottom-most 256 data memory locations, which implies that the blocks cannot be nested more than 256 levels deep. The memory map of the data memory is shown in Figure 8. The data words are stored in 32 bit locations and an instruction word is 16 bits.

There are six functional units in the implementation of this machine. They are:

1. The Arithmetic and Logic Unit
2. The Data Memory
3. The Instruction Processing Unit
4. The Micro-Instruction Processor
5. The Stack Control Unit
6. The Input/Output Processor

All the units are connected via a central bus. A block diagram of the system is shown in Figure 9. The various blocks in the diagram are described in the following sections.

The Arithmetic and Logic Unit (ALU) performs all the numeric and logic calculations. This unit also contains four registers which will hold the top four elements of the stack. Having the top elements in registers in the ALU eliminates a memory fetch cycle for most operand references.

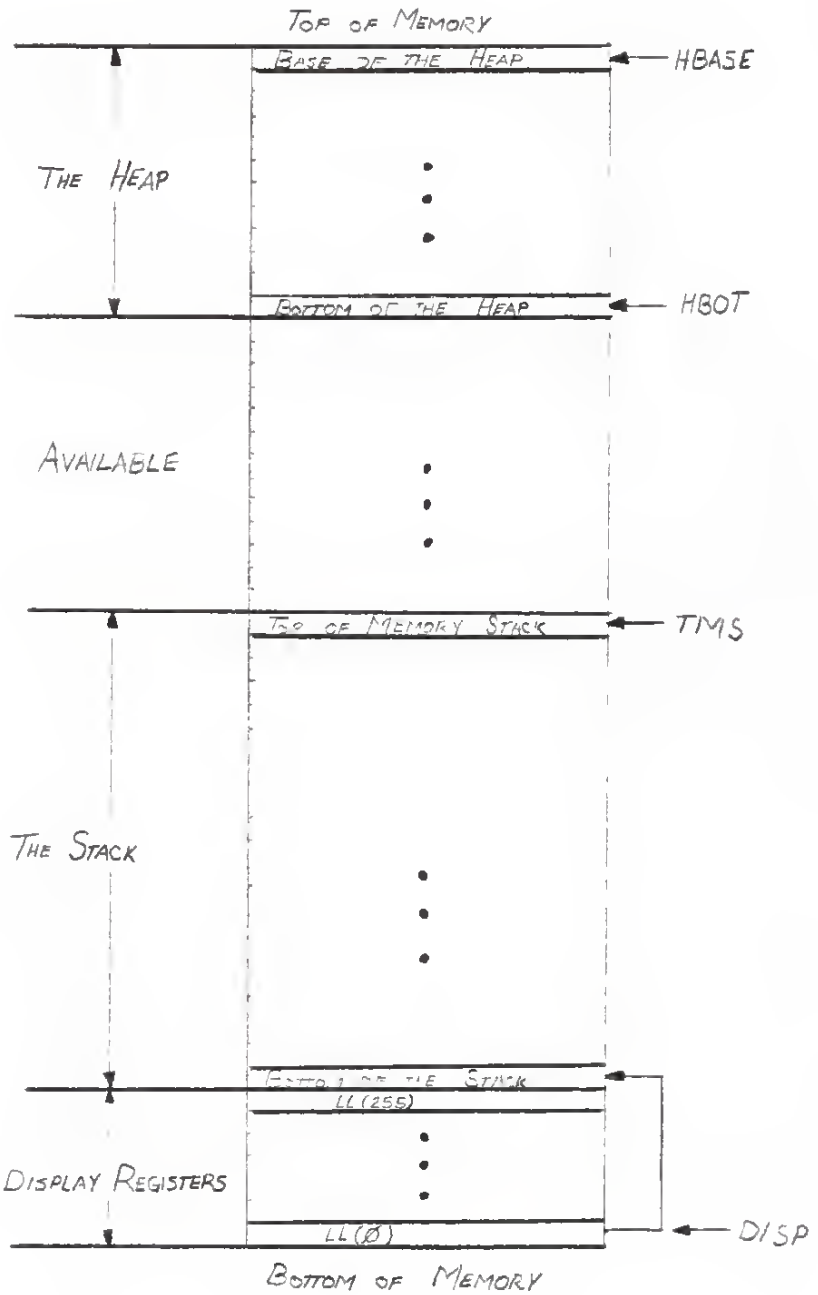


FIGURE 8. DATA MEMORY MAP

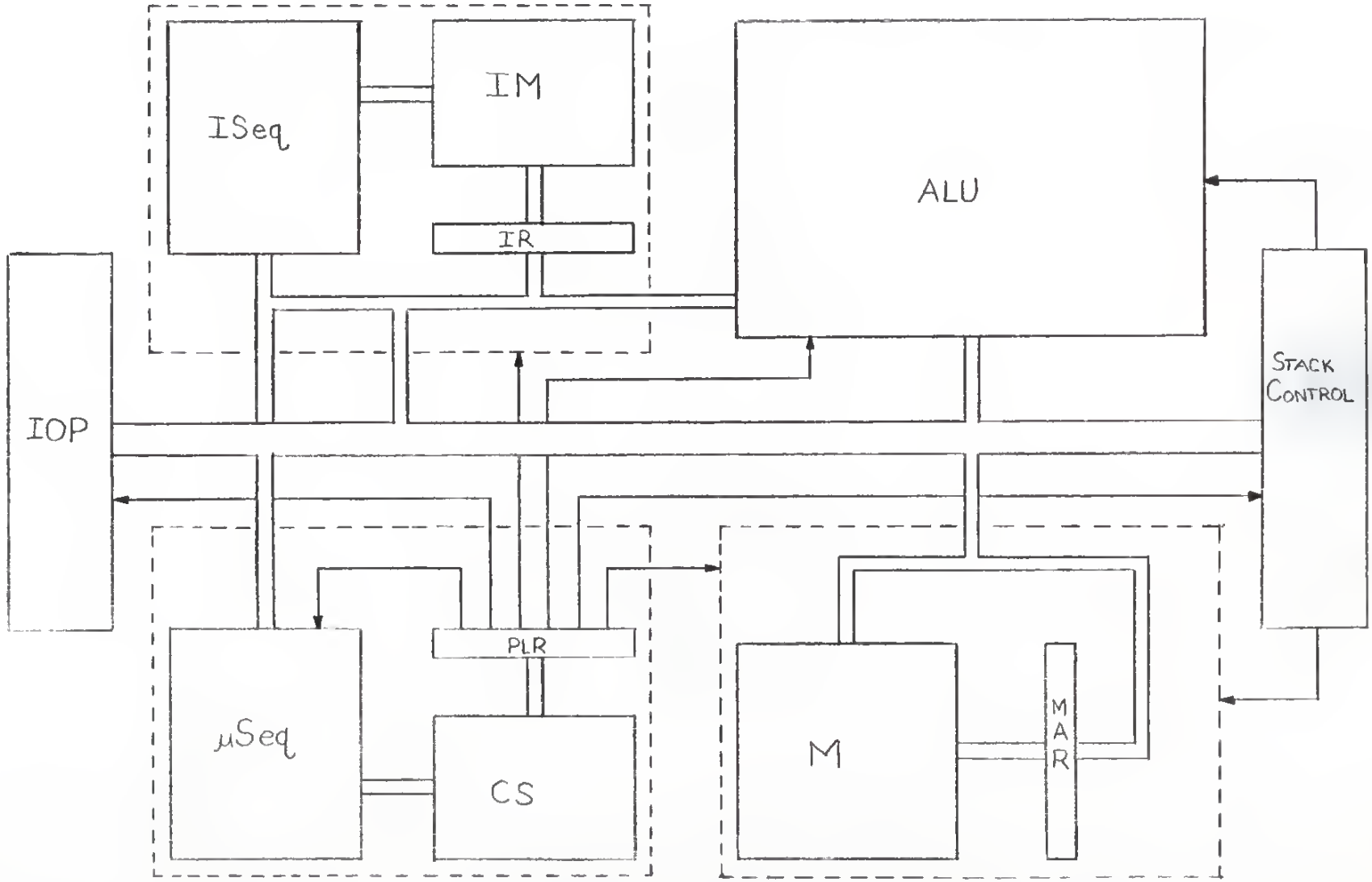


FIGURE 9. THE SYSTEM BLOCK DIAGRAM

The Data memory is used to hold both the stack and the heap. The memory unit includes the memory and the Memory Address Register. There is a possible address space of 4 Gwords.

The Instruction Processing Unit (IPU) is made up of three parts; the Instruction Sequencer (ISeq), Instruction Memory (IM), and the Instruction Register (IR). The Instruction Sequencer includes the conventional Instruction or program Counter. It updates the Instruction Counter to point to the next instruction as the current instruction is being fetched. It loads a new value into the IC for the jumps. The Instruction Memory contains the instructions. It can only be written to by the Console when the machine is halted. The third part of the IPU is the Instruction Register. This holds a copy of the currently executing instruction so that the next instruction can be fetched ahead. Figure 10 shows a block diagram of the IPU.

The Micro-Instruction Processor (uIP) controls the execution of the micro-instructions. There is a sequencing unit (uSeq) that generates the next micro-instruction address. There is a memory, the Control Store (CS), that contains the micro-instructions. And there is a Pipeline Register (PLR) to hold the current micro-instruction while the next is being fetched. The Pipeline Register sends the control signals to the other units. A block diagram of the Micro-Instruction Unit is shown in Figure 11.

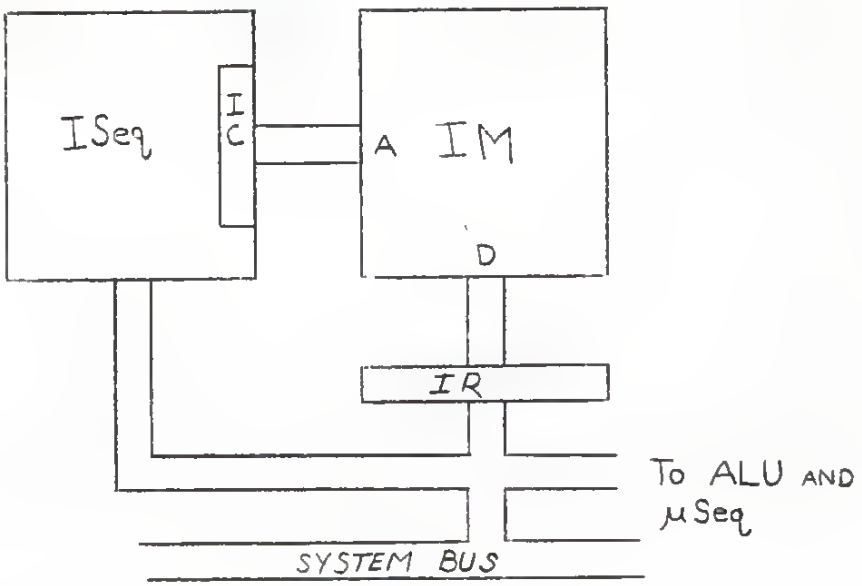


Figure 10. The Instruction Processing Unit

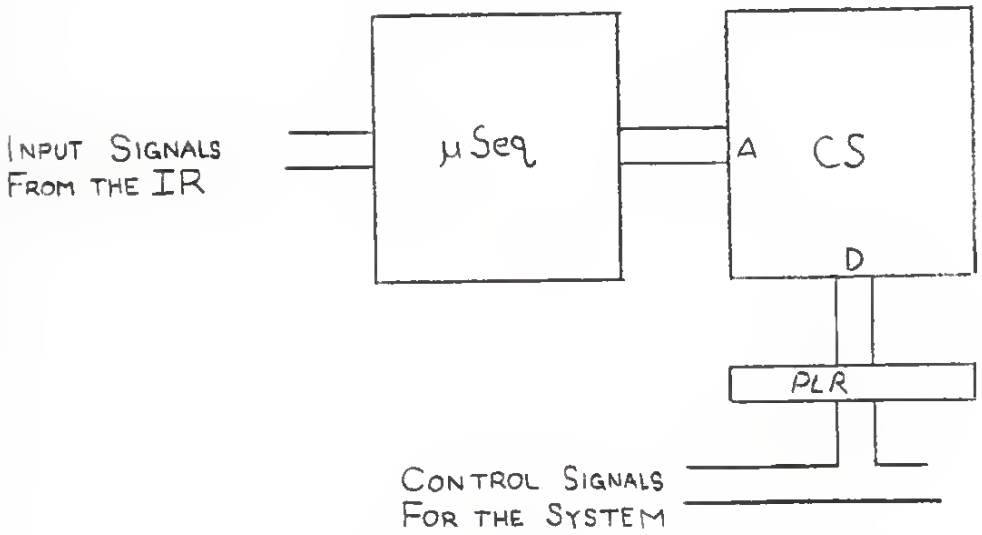


Figure 11. The Micro-Instruction Unit

The Stack Controller maintains the pointers necessary to implement the stack in memory and the counters needed to maintain the fast operand stack in the ALU. The pointers required for the memory stack are: The Top of the Memory Stack (TMS), The Base of the Display Registers (DISP), The Current Display Register Index (LLTOP), and the Bottom of the Heap (HBOT). The counters required for the fast stack are: The Number in the Fast Stack (NFS), the Top of the Fast Stack (TFS), and the Bottom of the Fast Stack (BFS). The Stack Controller also performs all the calculations on these pointers. The Stack Controller shares some hardware with the ALU.

The Input/Output Processor (IOP) has two major functions. It handles all I/O for the machine and it provides the Console interface. The Console is the master controlling device. It takes control by "HALTING" the processor. While the processor is in the HALT state, the Console can access any of the three memories. It can load the program, initial data, and the micro-code, or read any of them when necessary. The Console can also force the execution of a single micro-instruction or set the contents of either instruction register. When the machine is not halted, the Console is isolated from the rest of the machine. The IOP is implemented with a micro-processor to allow the most flexibility. A block diagram of the IOP is shown in Figure 12.

The system is synchronized by a clock signal (PHI). PHI

has a fifty-percent duty cycle. A clock cycle is considered to run from rising edge to rising edge. Figure 13 shows PHI.

The instruction registers latch in their new instructions on the rising edge of PHI. Both the macro-instructions and the micro-instructions are pipelined. This allows the next instruction to be fetched and waiting at the inputs of the respective instruction register while the current instruction is being executed. A typical sequence is shown in Figure 14. For this example, the micro-instruction sequence to perform the macro-instruction is only one micro-instruction long. Macro-instruction is abbreviated MI, and micro-instruction is abbreviated uI.

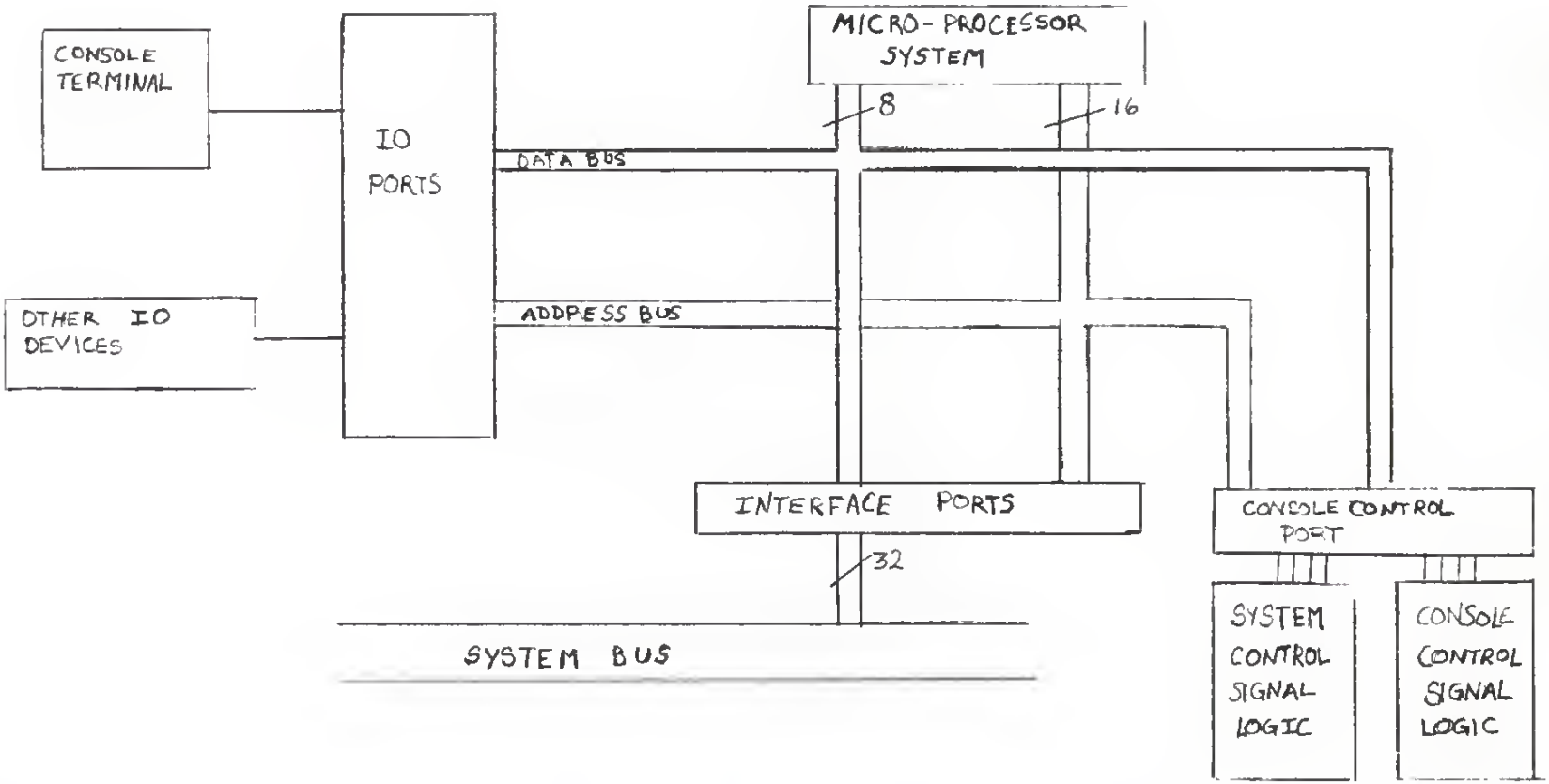


FIGURE 12. THE IOP BLOCK DIAGRAM

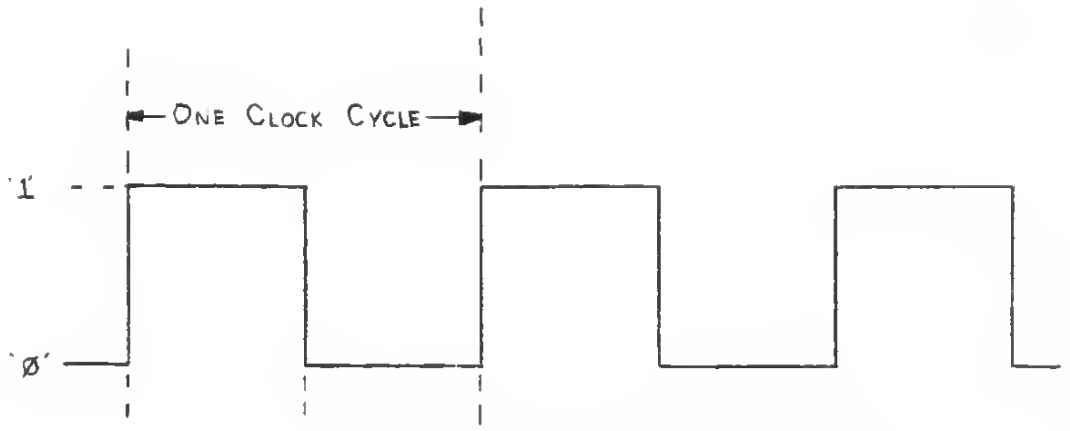


Figure 13. The System Clock, PHI

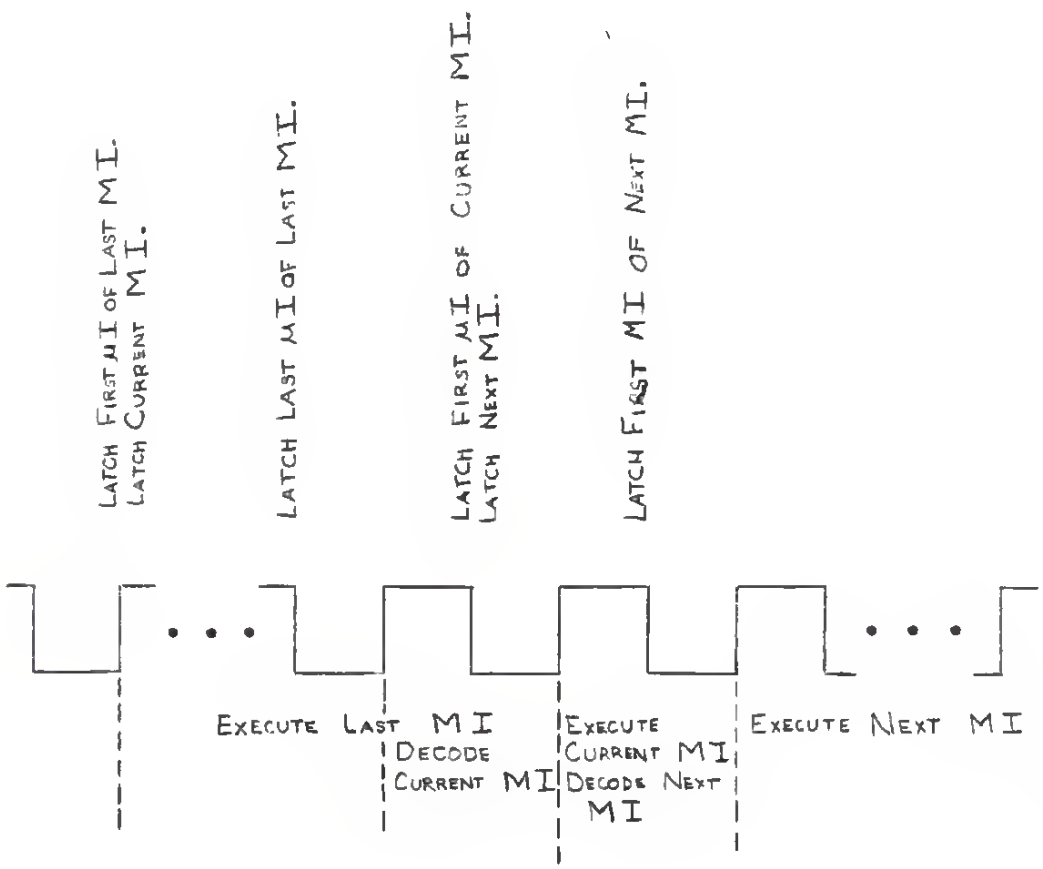


Figure 14. Simple Timing Example.

VI. THE ARITHMETIC AND LOGIC UNIT

The Arithmetic and Logic Unit (ALU) performs all the arithmetic and logic calculations. It is built around eight Advanced Micro Devices' AMD2901 four-bit slices. The eight slices are cascaded to form a 32 bit word length.

The 2901 contains 16 on-board registers. Four of these are used for a fast operand stack. If both operands for an operation are in the fast stack, the operation can be performed in one clock cycle. When the fast stack is full and a new operand is to be pushed, the oldest element in the fast stack is written out to the stack in memory, then the new operand is written into the just vacated location in the fast stack. The pointers necessary to implement the fast stack will be discussed in the chapter on the stack controller.

The ALU can have two sources of operands besides the internal fast stack. The operand may come from memory, which is the case if there are not enough operands in the fast stack for the operation or if a variable is read into the stack. The operand may also come from the instruction stream. The LOAD CONSTANT instruction, for instance, causes the next word in the instruction stream to be loaded onto the top of the stack. These two sources are multiplexed onto the D inputs of the 2901 array. The instruction word is only 16 bits wide so the high 16 bits are set to zero. A detailed block diagram of the ALU is shown in Figure 15.

The Exclusive-OR of the sign bit and the overflow flag, as well as the inverter on Q0 and the multiplexer on I1 are all used to implement Booth's multiplication algoithm (5). This allows multiplication in 34 clock cycles for a 32 bit operand.

Carry lookahead is provided across the slices by two 74182's, which have a third lookahead unit across them.

Most of the control inputs to the ALU are provided by the micro-instruction, so the design of this unit is reletively simple. The signals from the PLR are:

I0 - I2	Select the ALU operands	(3)
I3 - I5	Select the ALU operation	(3)
I6 - I8	Select the result destination	(3)
Cn	Determine the carry into the ALU	(1)
ASEL	Select the A register	(4)
BSEL	Select the B register	(4)
DSEL	Select the source of the D inputs	(1)
MULT	Signal a multiply operation	(1)
ALUOE*	ALU output enabled onto System Bus	(1)

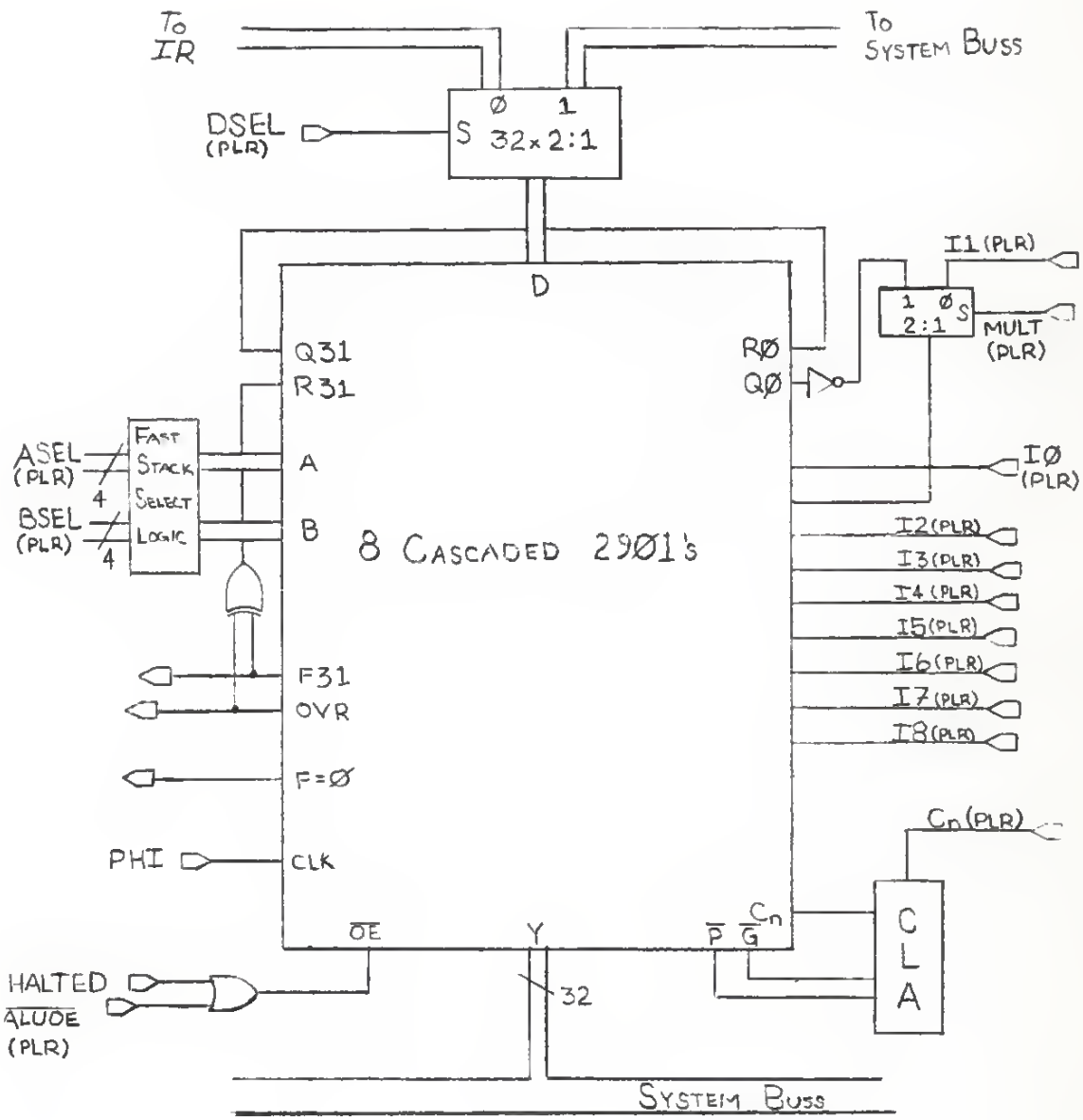


FIGURE 15. THE ALU BLOCK DIAGRAM

VII. MAIN MEMORY

The main memory contains the memory stack and the heap. Thirty-two address lines are available to give a theoretical address space of four gigawords. Many implementations of the memory are possible with variations depending on the type of memory chips and the decoders used. One of the possible implementations is presented as an example. The implementation that is presented uses the HM6116 2K X 8 CMOS static RAM chip, and 4-to-16 decoders. Four of the memory chips are paralleled to form a 32 bit word length.

The Memory Address Register (MAR) is 24 bits wide. This could easily be expanded to 32 bits. The 24 bit address provides 16M words. A11, A12, and A13 are taken as inputs to a 74154 4-to-16 decoder. The '154 has two active low enables. MEMSEL*, from the PLR, is tied to one of the enables. MEMSEL* is the active low signal indicating that the memory is to have control of the bus for the next clock cycle. The other enable can be used as the select signal from a higher decode level. Each output from the decoder will select one of sixteen blocks of four memory chips. Each section of 16 blocks (the amount decoded by one decoder) is called a bank. If a second level of decode were provided, then the 0000 output from the second level would enable the BANK0 decoder, which in turn would select one of sixteen blocks. The lowest level decoder and associated memory chips would be repeated sixteen times for each BANK decoder.

A write operation should occur during the last half of the clock cycle (while PHI is low). This gives all the units that might supply the data to the memory time to settle down. The WRITE ENABLE (WREN*) signal to the memory chips is generated by ORing the PLR signal WRITE* with the system clock, PHI.

The block diagram of the memory is shown in Figure 16. Figure 16 assumes only one level of decoding is provided so the second enable input is tied low.

The signals required from the PLR are;

MEMSEL*	Memory has the bus for the next clock cycle
WRITE*	Memory access is a write.
ENMAR*	Enable the Memory Address Register.

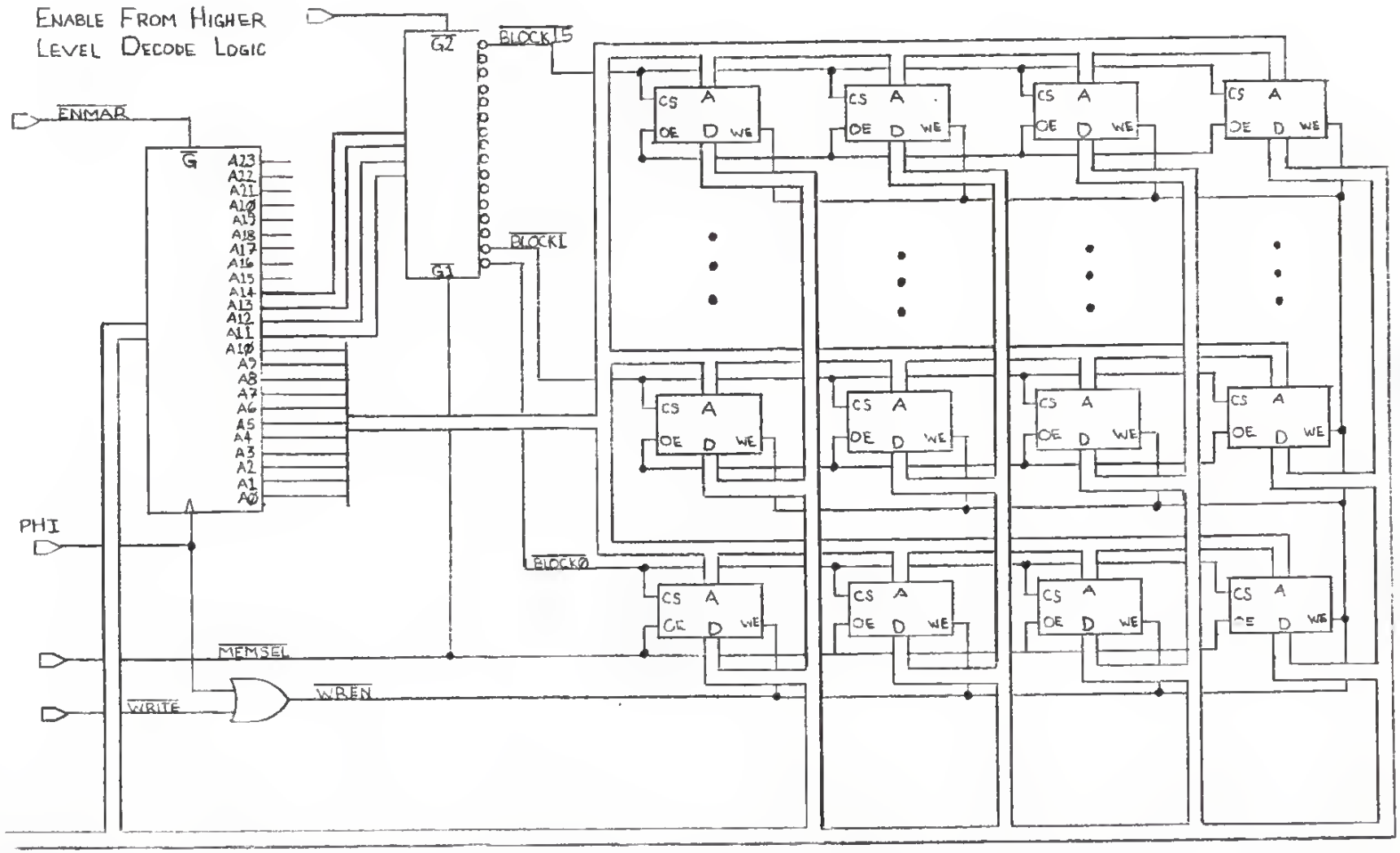


FIGURE 16. THE MAIN MEMORY BLOCK DIAGRAM

VIII. INSTRUCTION PROCESSING UNIT

The Instruction Processing Unit (IPU) handles the fetching of the macro-instructions. The IPU has three major parts. It contains a sequencing unit (ISeq) to handle the generation of the next address, an instruction memory (IM), and an Instruction Register (IR). By having a processor and memory exclusively for the IPU, instruction fetches can be carried out in parallel with the execution of instructions. The instruction which is being executed is stored in the Instruction Register while the next instruction is being fetched. The synchronization of address generation, instruction fetching, and loading of the Instruction Register is controlled by the micro-instruction. Once the micro-sequencing unit has decoded the macro-instruction, the next macro-instruction can be loaded into the IR. Thus the micro-instruction which causes a jump to the routine to implement the macro-instruction includes the signals to fetch the next macro-instruction and load it into the IR. The Instruction Memory is Read/Write Memory, but it appears as ROM during the execution of a program. The Console is the only device that can write to the IM. The IPU contains the necessary logic to allow the Console to access the IM over the system bus. The instructions words are sixteen bits so the Instruction Memory is sixteen bits wide.

The Instruction Sequencer generates the address of the next macro-instruction. The are only two ways that a next

address can be determined by this machine architecture. The most common method is sequential addressing, where the next instruction is the one immediately following the current one. The other method, used by the jump instructions, fetches the next instruction from an address that has been loaded from the system bus. There is no relative branching in this machine. Signals provided by the micro-instruction to perform these two types of next address generation are NEXTI and JUMP.

The Instruction Sequencer is implemented with four 2901 bit slices. Register 0 is used as the Instruction Counter (IC). In sequential addressing the contents of the IC are routed to the output of the 2901's to be used as the instruction address. At the same time, the IC is incremented and on the rising edge of the system clock the incremented value is stored back into the IC. When a JUMP is executed, the IC latches the value from the D inputs of the 2901's. This value is also passed through the 2901's ALU to the Instruction Memory.

Figure 17 shows the IPU. The two control signals JUMP and NEXTI from the PLR are mapped to the control inputs (I0 - I8 and Cn) of the Instruction Sequencer to provide the increment and load operations as shown in Table 5. A jump can be either conditional or unconditional. A conditional jump is made if the word at the top of the stack is FALSE. FALSE is defined as all ones and TRUE is defined as all zeros. The F=0 flag from the ALU indicates the top word was TRUE so (F=0)* is used to determine if the jump is made. The unconditional jump

is generated in the same way as the conditional jump except the condition is forced to pass by the PLR signal FORCE. The conditional signal is called GOJUMP. The idle state 'DO NOTHING' requires I0 and I1 be '0' so the signal from the OR is ANDed with JUMP to ensure that if both JUMP and NEXTI are low then nothing is done.

Condition	Cn	I8	I7	I6	I5	I4	I3	I2	I1	I0
Sequential	1	0	1	0	0	0	0	1	0	0
Conditional Jump (fail)	0	0	1	1	0	1	1	1	0	0
Conditional Jump (pass)	0	0	1	1	0	1	1	1	1	1
JUMP	0	0	1	1	0	1	1	1	1	1
Do Nothing	0	0	1	0	0	0	0	1	0	0

Cn = NEXTI
I8 = 0
I7 = 1
I6 = JUMP
I5 = 0
I4 = JUMP
I3 = JUMP
I2 = 1
I1 = GOJUMP
I0 = GOJUMP

Table 5. The Instruction Processor Unit Control Signals

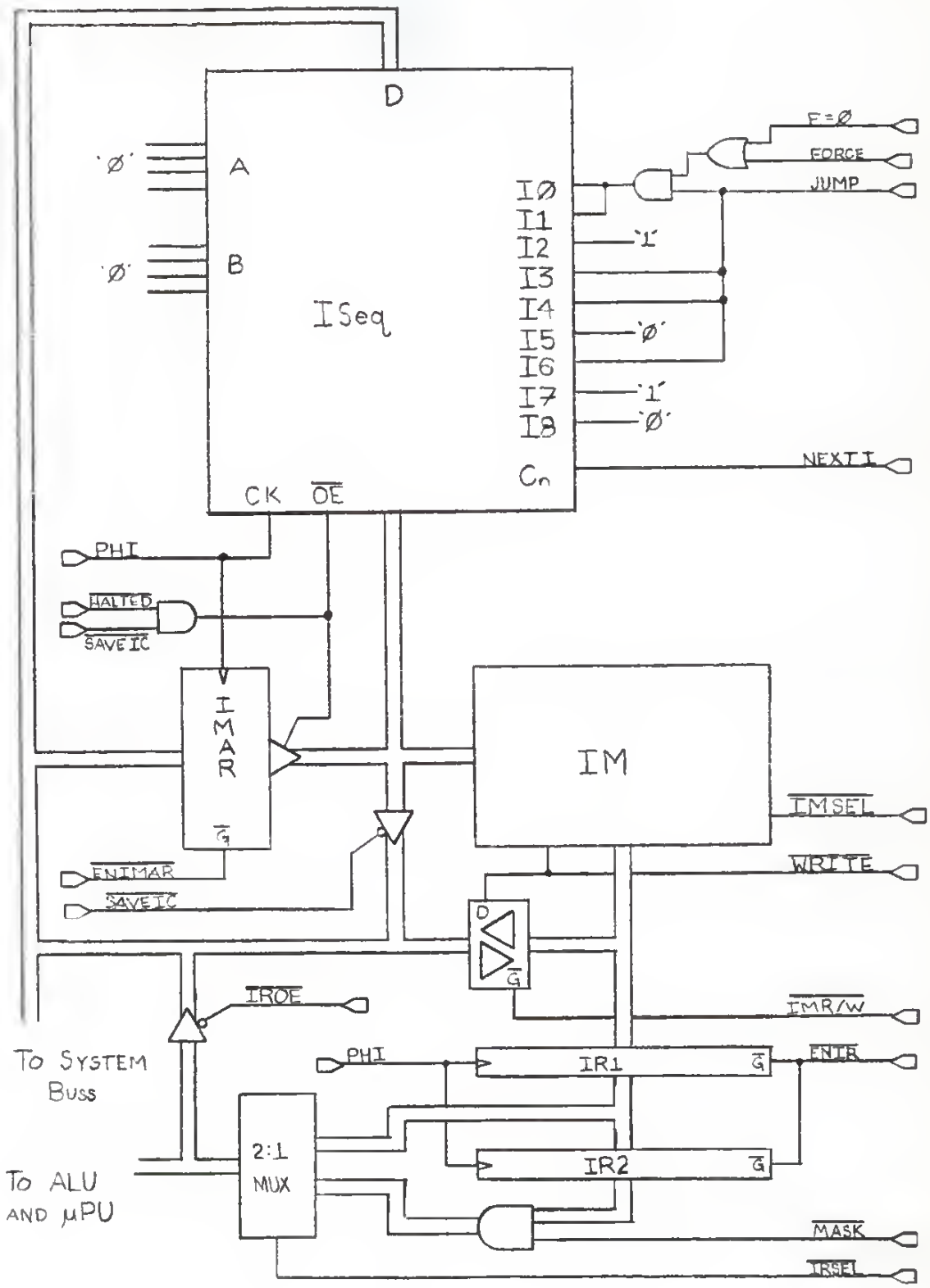


FIGURE 17. THE IPU BLOCK DIAGRAM

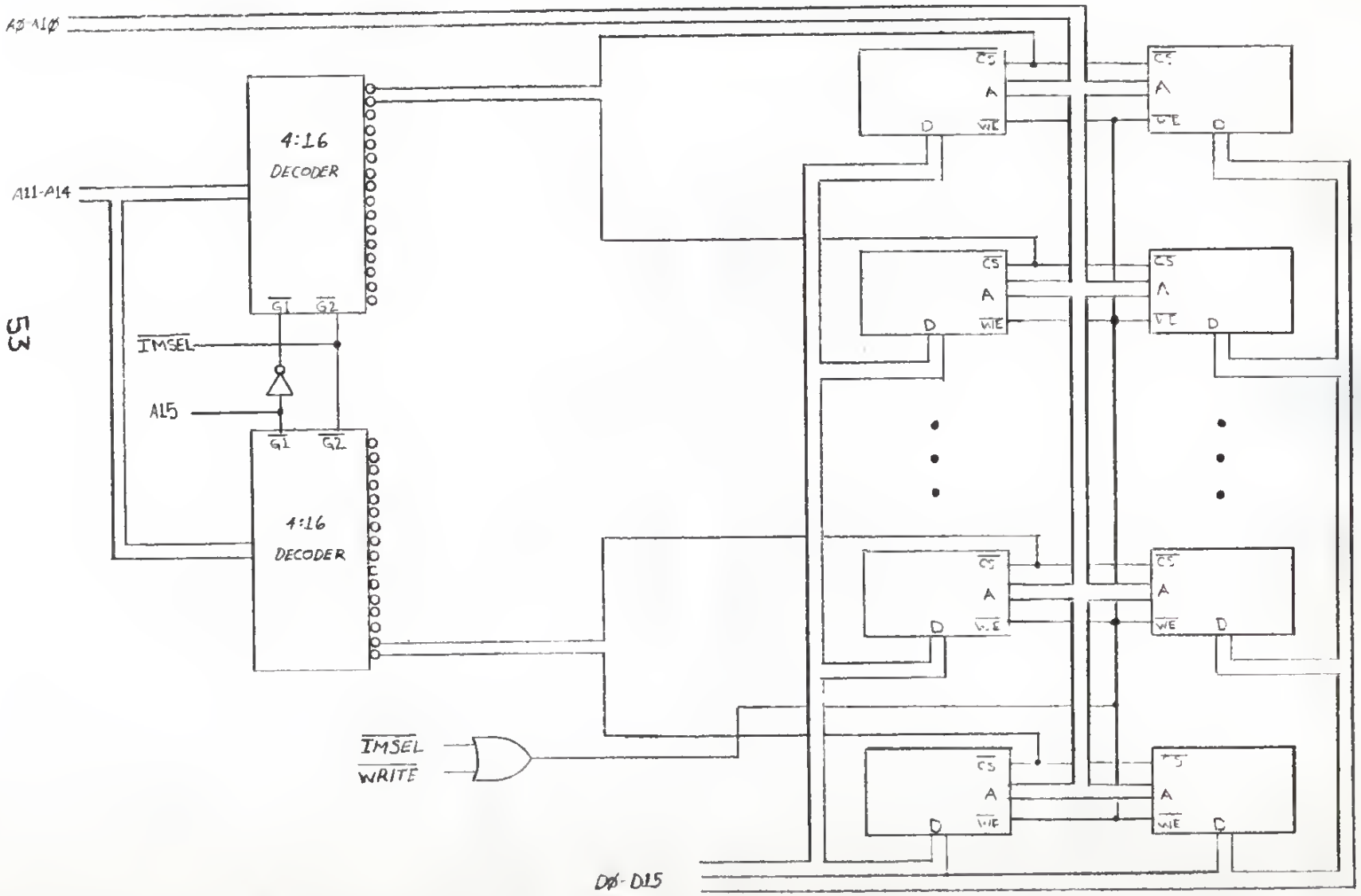


FIGURE 18. INSTRUCTION MEMORY BLOCK DIAGRAM

A procedure call requires that the return address be stored on the stack. Three-state gates interface the outputs of the Instruction Sequencer to the system bus. These gates are controlled by the signal SAVEIC* from the PLR.

The Instruction Memory (IM) implementation has many variations as did the Data Memory. Again, a possible implementation is presented as an example. The same 2K X 8 static RAM's are used in the Instruction Memory example as were used in the Data Memory. Figure 18 shows the Instruction Memory block diagram for this implementation. The 4-to-16 decoders each decode 32K words. The decoder chips are 74154's. A write to the IM is allowed only when the Console lowers the IMSEL* line and the WRITE* signal is given.

The pipelined architecture used in this machine introduces some complications. If the instruction mapping logic detects that there are not enough operands in the fast stack, then instead of executing the instruction, a micro-code "fix" routine is executed instead. When this routine is finished, the instruction execution can proceed. However, in the pipelined architecture, the instruction has already been over written in the IR. The same kind of problem arises when the lexical level is stored in the same instruction word as the op-code. By the time the micro-code routine has figured out that it needed the second half of that last instruction word, the word is gone. There are two solutions to this. The first is to delay fetching the next instruction until the micro-code

routine has determined that it no longer needs the current instruction. This means that the next macro-instruction can not be loaded into the IR until at least one micro-instruction has executed. If the micro-instruction routine was only one instruction long, the IR will be loaded at the end of that one instruction. Then another clock cycle must pass while the mapping logic decodes the new macro-instruction. No instruction could take less than two clock cycles. The other solution is to keep a copy of the IR. This copy is one step behind the IR. The same control signals cause the copy register (called IR2) to latch data as cause the IR to latch data, but its inputs are tied to the outputs of the IR. Then as the IR (henceforth called IR1) latches the new instruction, IR2 is latching the old one. The micro-code fix routines can choose to use the value stored in IR2 as the instruction to execute. The micro-code routine that evaluates the address double can get the lexical level from IR2. The output of IR2 is often used as the lexical level in a variable address evaluation. To simplfy this, a means of masking off the high 8 bits of the IR2 field is provided. This function is enabled by the signal MASK* from the PLR. The two IR's are multiplexed together. Which IR will be used for a particular clock cycle is selected by the IRSEL signal from the PLR. The output of the IR multiplexer might be needed by any of the functional units. The IOP, Memory, and IPU get the value from the bus. The ALU typically needs the value in the IR when it is calculating a variable address. The ALU can take the variable displacement

from the D inputs and add it to the display register that it has already gotten, and pass the result through the system bus to be loaded into the Memory Address Register all in one clock cycle. If the ALU had to get the displacement value from the system bus, then the above operation would require two cycles, one for the IR to put the displacement on the bus, and the other for the ALU to put its result on the bus to be latched into the MAR. For this reason, there is a separate path from the output of the IR multiplexer to the ALU D input multiplexer. For similar reasons there is a separate path from the multiplexer to the mapping logic in the micro-processing unit. The three-state gates that allow the IR onto the bus, are activated by the PLR signal IROE*.

The Console is able to read or write to any memory location in the Instruction Memory. It is also able to load a value into either the IC or the IR, and to read the IC. In order to access the memory, an Instruction Memory Address Register must be provided. This IMAR gets its data from the system bus and is loaded when the signal ENIMAR* from the Console is TRUE (0) and there is a rising edge on the clock. The data into or out of the IM also comes from the bus. This requires bus transceivers between the IM and the bus. The transceivers are enabled by the signal IMR/W*. The PLR control signals that load the IC and the IR are controlled by the Console when the machine is halted. This interface takes place at the PLR so it does not show up in the IPU circuit. The

outputs of the 2901's must be disabled when the IMAR is used. For the Console to read the IC the 2901's must be enabled again and the IMAR output must be three-stated. The three-state gates that allow the IC onto the bus must also be opened. The SAVEIC* signal from the PLR is controlled by the Console but it is ANDed with HALTED so that the ISeq only has its outputs enabled when the machine is running or when it is halted and the IC is being read.

The Instruction Processing Unit requires eight signals from the PLR. They are:

JUMP	-	load the IC from the system bus
FORCE	-	force the jump to be taken
NEXTI	-	increment the IC
ENIR*	-	enable the Instruction Register
SAVEIC*	-	allow the IC onto the system bus
IRSEL	-	select between IR1 and IR2
MASK*	-	mask off the high 8 bits of IR2
IROE*	-	allow the IR onto the system bus

IX. THE MICRO-PROCESSING UNIT

The micro-processing unit (uPU) controls the fetching of the micro-instructions. The micro-instruction is 64 bits wide. It has several fields which provide control signals to the different units within the machine including the micro-processing unit. The micro-processing unit is organized much like the Instruction Processing Unit. It has three major components consisting of a sequencing unit (uSeq), a memory, and a register to allow fetch ahead. The sequencer generates the address of the next micro-instruction. The micro-instruction is fetched from the Control Store (CS), and latched into the Pipeline Register (PLR) on the rising edge of the system clock.

The micro-sequencer is built around the AM2910 micro-program controller. A block diagram of the entire micro-sequencing unit is shown in Figure 19.

The 2910 can use the D inputs as one source of the next address. It provides three signals for enabling one of three sources onto the D inputs. One of these sources is the PLR. The enabling signal is called PL*. Twelve bits of the PLR are routed to the D inputs to be used for such things as a micro-instruction branch address or a counter value. Another source, called MAP, is enabled by the MAP* signal. Typically, MAP enables some kind of mapping PROM or PLA that maps the macro-instruction op-code into a micro-code entry point. In the

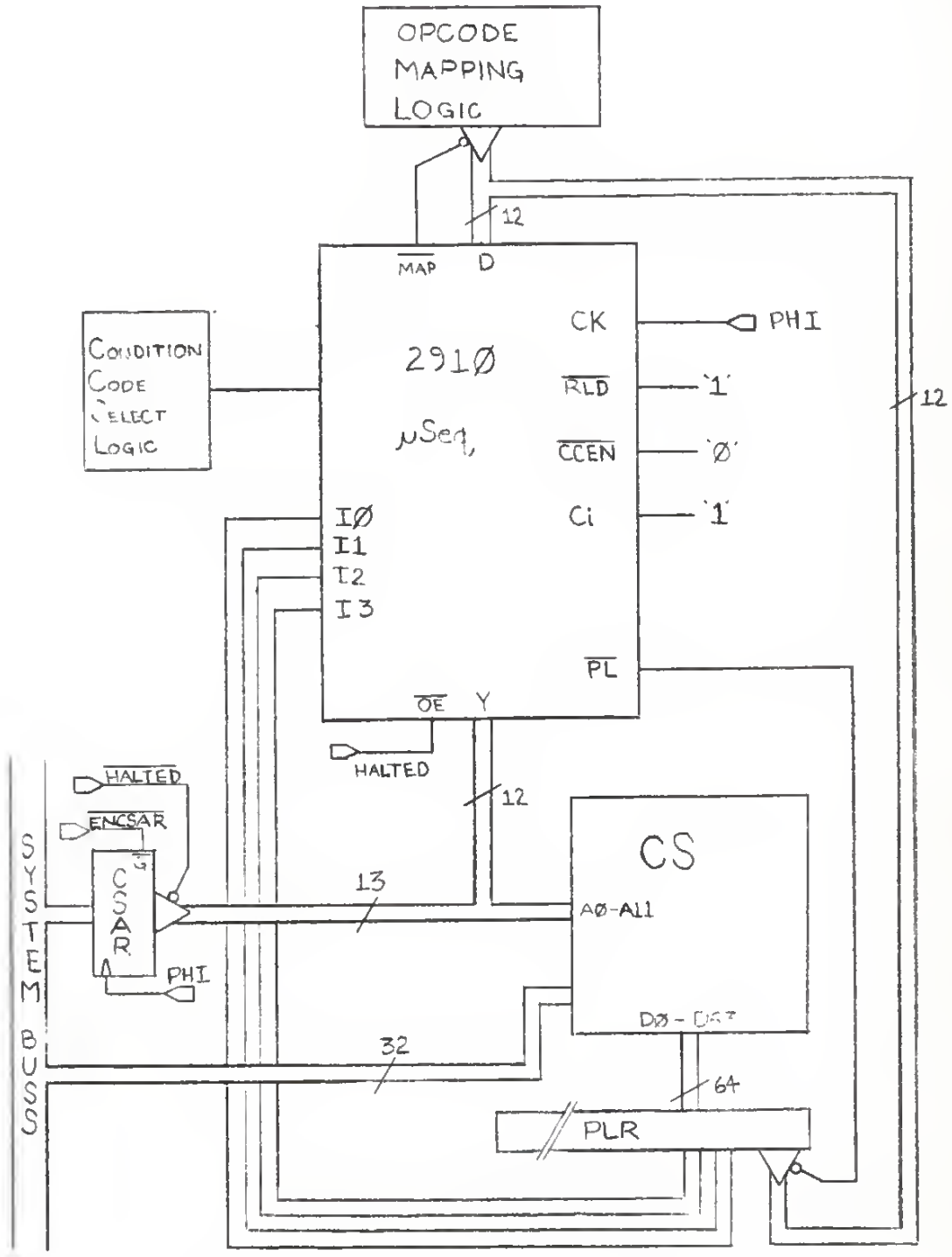


FIGURE 19. MICRO-PROCESSING UNIT BLOCK DIAGRAM

machine, some instructions require a certain number of operands on the fast operand stack in the ALU registers. The mapping logic includes a test on the number of operands and generates the address of a "fix" routine if there are not enough or too many. There are three fix routines. One is used to bring an operand into the fast stack from the memory stack if there are an insufficient number of operands to perform the operation. The second fix routine writes the oldest element in the fast stack out to the top of the memory stack if there is not enough space in the fast stack for an new operand to be written. The third fix routine empties the fast stack. This is used before changing context such as in a procedure call. Figure 20 shows a diagram of the mapping logic.

The number of operands needed is given in the first three bits of the macro-instruction op-code. The three bit field is translated as:

I0	I1	I2	Operands needed	
0	0	0	NEED0	
0	0	1	NEED1	
0	1	0	NEED2	
1	0	0	NEED-1	(Need an empty location)
1	1	1	FLUSH	(Empty the fast stack)

The number of operands available is known from the NFS counter. The number needed and the number available are decoded to give a signal corresponding to each of the possibilities. The combinations that require an operand be brought into the fast stack are:

NEED1 and HAVE0
NEED2 and HAVE0
NEED2 and HAVE1

The combination that requires an operand to be written out to memory is:

NEED-1 and HAVE4

The combination that requires the stack be emptied is :

FLUSH and NOT HAVE0

The pipeline nature of the machine allows the new instruction to be decoded while the previous instruction is being executed. If the previous instruction changes the fast stack counters, the change does not take effect until the end of the clock cycle, that is, after the decoding has finished and when the new instruction's first micro-instruction is being latched. Thus the mapping logic also needs to detect whether the NFS will change at the end of the clock cycle. The signals PLUS1 and MINUS1 indicate what the NFS will do. PLUS1 is TRUE when the fast stack will increase in depth and MINUS1 indicates that it will decrease. Combining these signals with the signals derived from the number needed and number available yields the following combinations:

Combinations requiring an operand to be read

HAVE0 and NEED1 and NOT PLUS1
HAVE0 and NEED2
HAVE1 and NEED1 and MINUS1
HAVE1 and NEED2 and NOT PLUS1
HAVE2 and NEED2 and MINUS1

Combinations that require a location to be emptied

HAVE3 and NEED-1 and PLUS1
HAVE4 and NEED-1 and NOT MINUS1

Combinations that require the fast stack be flushed

NOT EMPTY and FLUSH

where EMPTY is

HAVE0 and NOT PLUS1 or HAVE1 and MINUS1

If any of the conditions is detected, the corresponding input to a priority encoder is pulled to 0. The output of the encoder then selectes one of four inputs to a multiplexer. If none of the conditions is TRUE, the selected operation may proceed. The fourth input to the priority encoder is tied to zero so that if none of the first three is low, the op-code is allowed to select the micro-instruction jump address.

The third source for the D inputs is enabled by the signal called VECT*. VECT* is usually used for jumping to an interrupt service routine. Interrupts are not supported in this implementation. Interrupt decode logic could provide some sophisticated features such as PL/1's "On" conditions, but this is beyond the scope of this paper.

The 2910 has two conditional inputs. These are used by many of the 2910's instructions to determine whether a conditional branch is taken. CC* is one of the conditional inputs. If it is low, the conditional branch will occur. If high, the next micro-instruction will be fetched. The other conditional input is CCEN*. When this is low, testing of CC* is as described above. When CCEN* is high, CC* is unconditionally TRUE (0). CCEN* is tied low in this implementation. A branch can be forced by setting CC* = '1'.

The CC* input is the output of a 16-to-1 multiplexer. A four-bit field from the PLR selects from one of sixteen inputs including six conditional signals, the signal EMPTY, indicating the fast stack is empty, the two I/O handshaking signals, or the FORCE input. Figure 21 shows the generation of CC*.

The Control Store is the memory that holds the micro-instructions. Its address inputs are provided by the 2910 and its data outputs go to the PLR. However the Console must have access to both the address and the data lines. A Control Store Address Register (CSAR) and bus transceivers are provided to interface the CS to the system bus so that the Console can access the CS. The CSAR is enabled by the ENCSAR* signal from the Console. The bus transceivers are enabled by the signal CSR/W*, also from the Console.

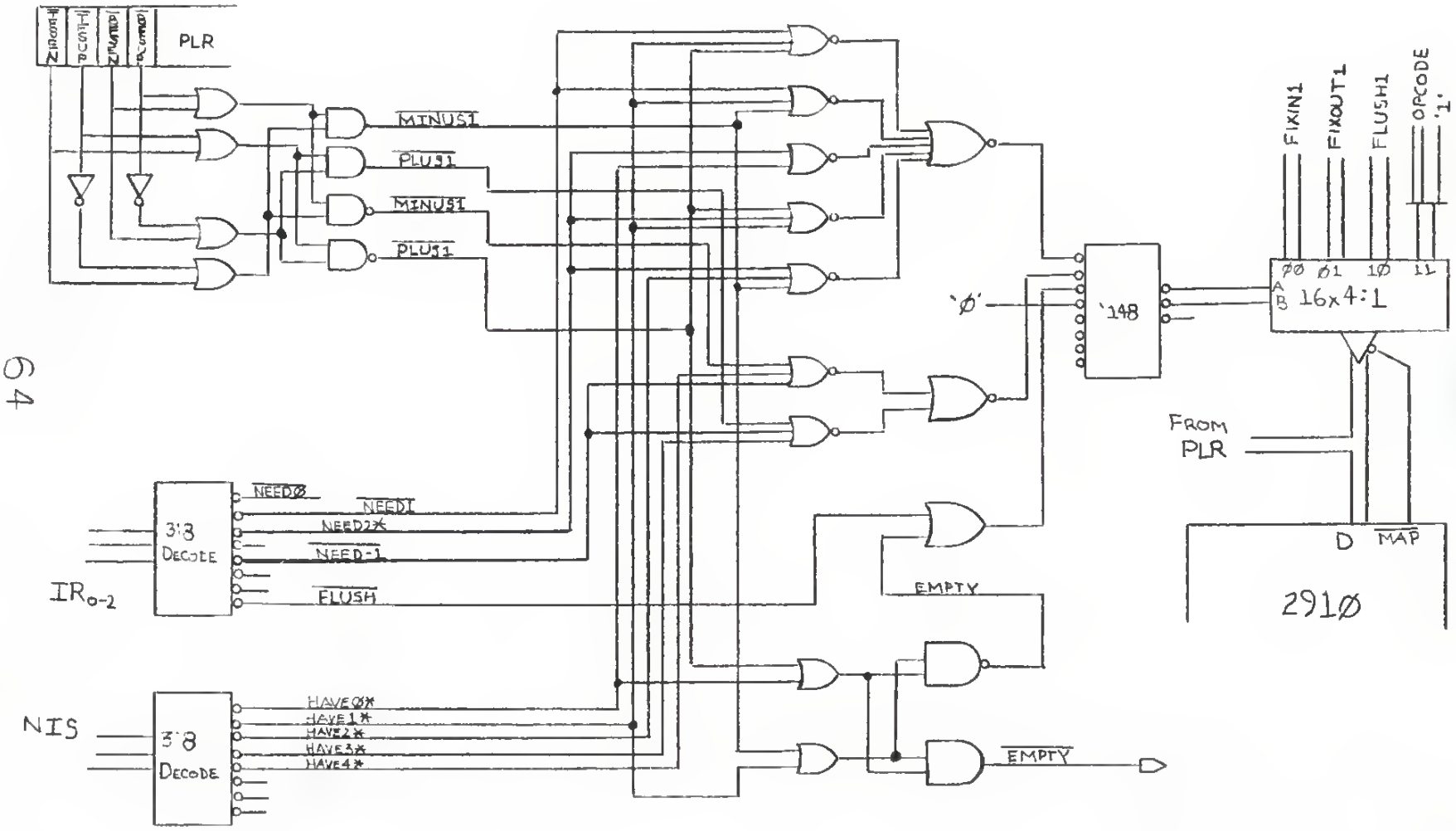


FIGURE 20. THE OPCODE MAPPING LOGIC

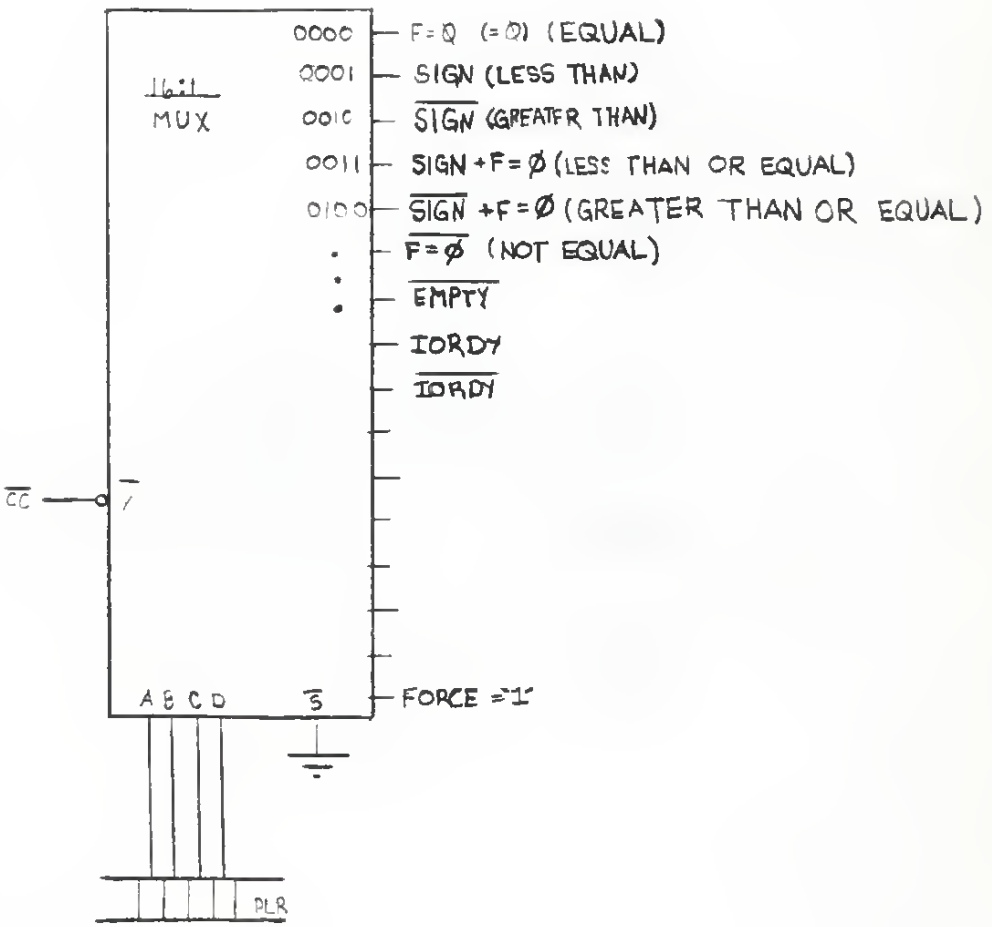


FIGURE 21. GENERATION OF THE $\overline{CC}$ SIGNAL

One complication arises in that the micro-word is wider than the system bus. The Console accesses half of the CS at one time. Address A1 from the CSAR is tied to A0 of the CS. Then A0 of the CSAR is used to select between the upper or lower 32 bits of the micro-word. The control store is shown in Figure 22. The lower half of the bus transceivers are enabled by the signal CSR/W*, and the upper half by CSR/W*.

The PLR is 64 bits wide. It is shown in Figure 23 with each of the bits defined. The 12 bits that are used as the 2910 branch address have three-state outputs, which are enabled by PL* from the 2910. The bits in the PLR should be zero whenever possible for the "Do Nothing" case. Several of the bits are inverted so that they can be used as negative TRUE signals but can be programmed as positive TRUE signals. The Console can load the PLR. The data comes from the system bus in 32 bit pieces. Two signals from the console determine which half of the PLR will be loaded. They are ENPLR0* to load the low half and ENPLR1* to load the high half. These are shown in Figure 22.

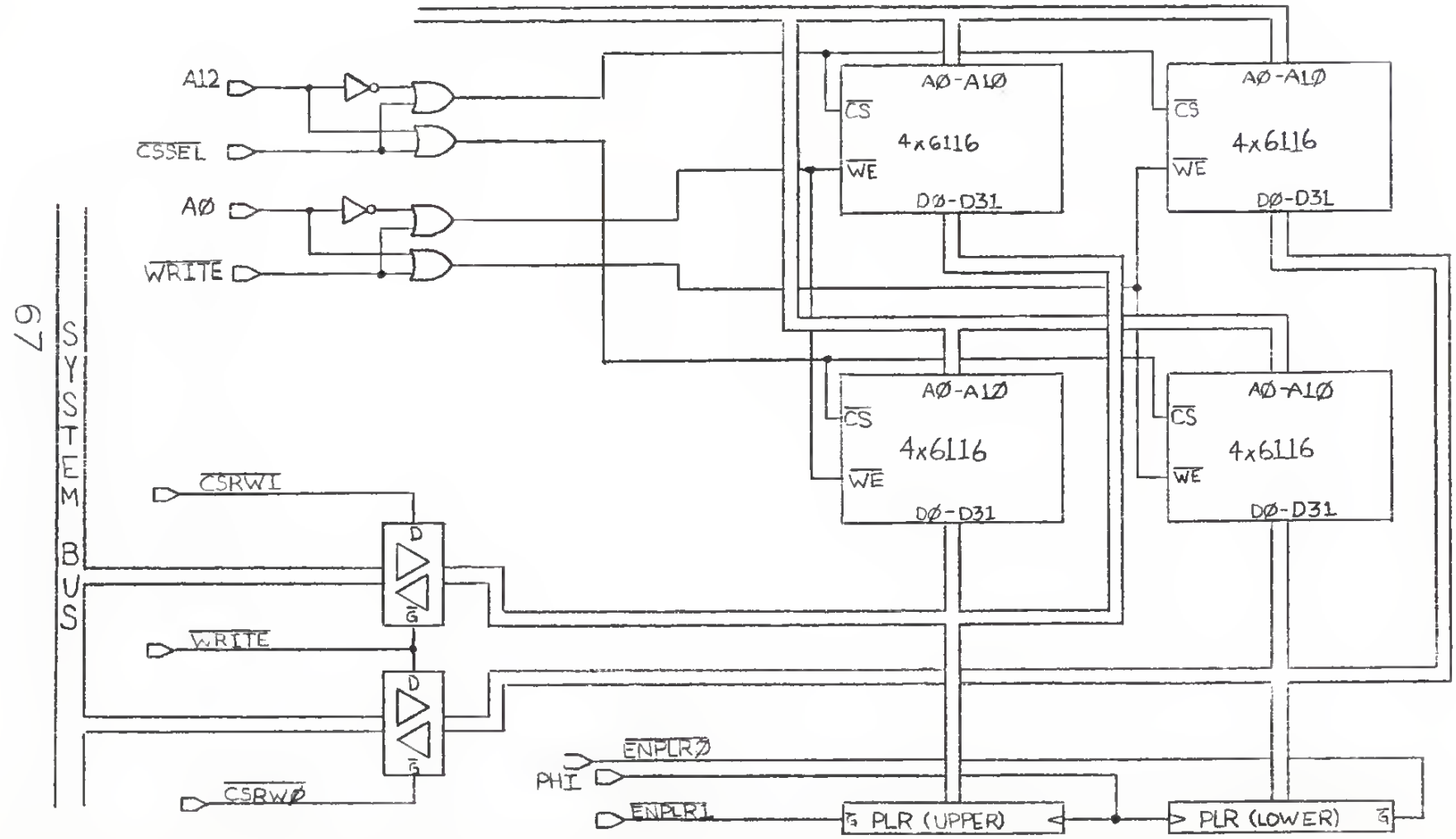


FIGURE 22. THE CONTROL STORE BLOCK DIAGRAM

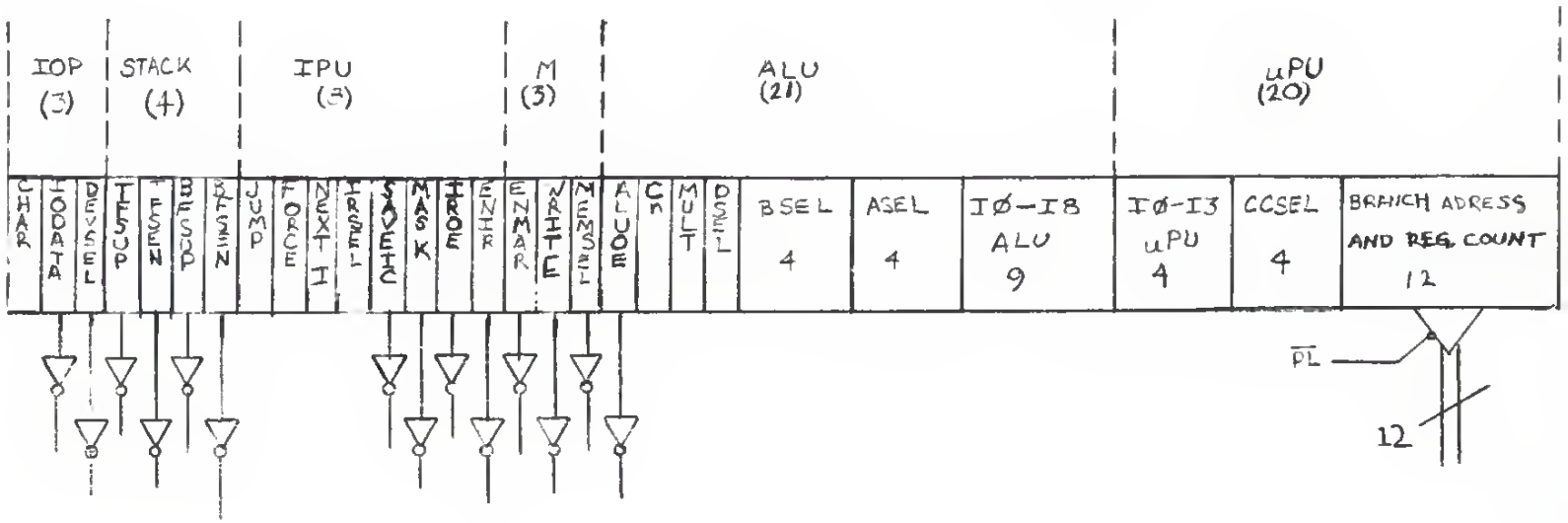


FIGURE 23. THE PIPELINE REGISTER

X. STACK CONTROLLER UNIT

The stack controller has two major functions. It controls the pointers and counters necessary to maintain the Fast Operand Stack in the ALU registers, and it maintains the pointers for the stack in memory.

The Fast Stack is treated as a circular stack. When it becomes full, the next element is written into the location of the oldest element. The Top of the Fast Stack (TFS) pointer points to the ALU register which will receive the next element. The TFS pointer is implemented with a 74191 four-bit up/down synchronous counter. The two most significant bits are ignored. This gives the circular addressing. The enable signal (TFSEN*) and the direction signal (TFSUP*) come from the PLR.

In order to preserve data when the stack becomes full, the oldest element must be written out to the stack in memory. This requires knowledge of the location of the oldest element or "Bottom of the Fast Stack". The Bottom of the Fast Stack is also the location into which the value from the memory stack is copied when there are not enough operands to perform an operation. The Bottom of the Fast Stack pointer (BFS) is used to point to this location. It always points to the least recently entered element. It is also implemented with a 74191 four-bit counter. The BFS counter is enabled by the signal BFSEN* from the PLR and the direction of count is controlled by

the signal BFSUP*, also from the PLR.

The PLR references the Fast Stack elements relative to the TFS or BFS pointers. The Fast Stack control logic maps these logical addresses to the actual ALU register addresses. Figure 24 shows how the mapping from the PLR's logical address to the actual ALU register address is performed. The subtraction is performed by a 74181 four-bit function generator. The control inputs of the function generator are tied such that a subtraction is performed (S3=0 , S2=1 , S1=1 , S0=0 , M=1 , and Cn=0). The unused data inputs are all tied to '0'. Table 6 shows how all the register select combinations from the PLR are mapped to the register select inputs of the ALU.

A third counter keeps track of the number of operands in the fast stack. This information is used by the uSeq unit to determine whether enough operands are in the fast stack to perform an operation. The value in the counter (called NFS) could be derived from the values in TFS and BFS, but since the path from the IR through the mapping logic and uSeq and then the Control Store fetch is potentially the limiting path for increasing the processor speed, a separate counter is kept. For the same reason, all components in the critical path should be Schottky TTL. The control signals for the NFS counter are not in the critical path so they are derived from the control signals for the TFS and BFS counters.

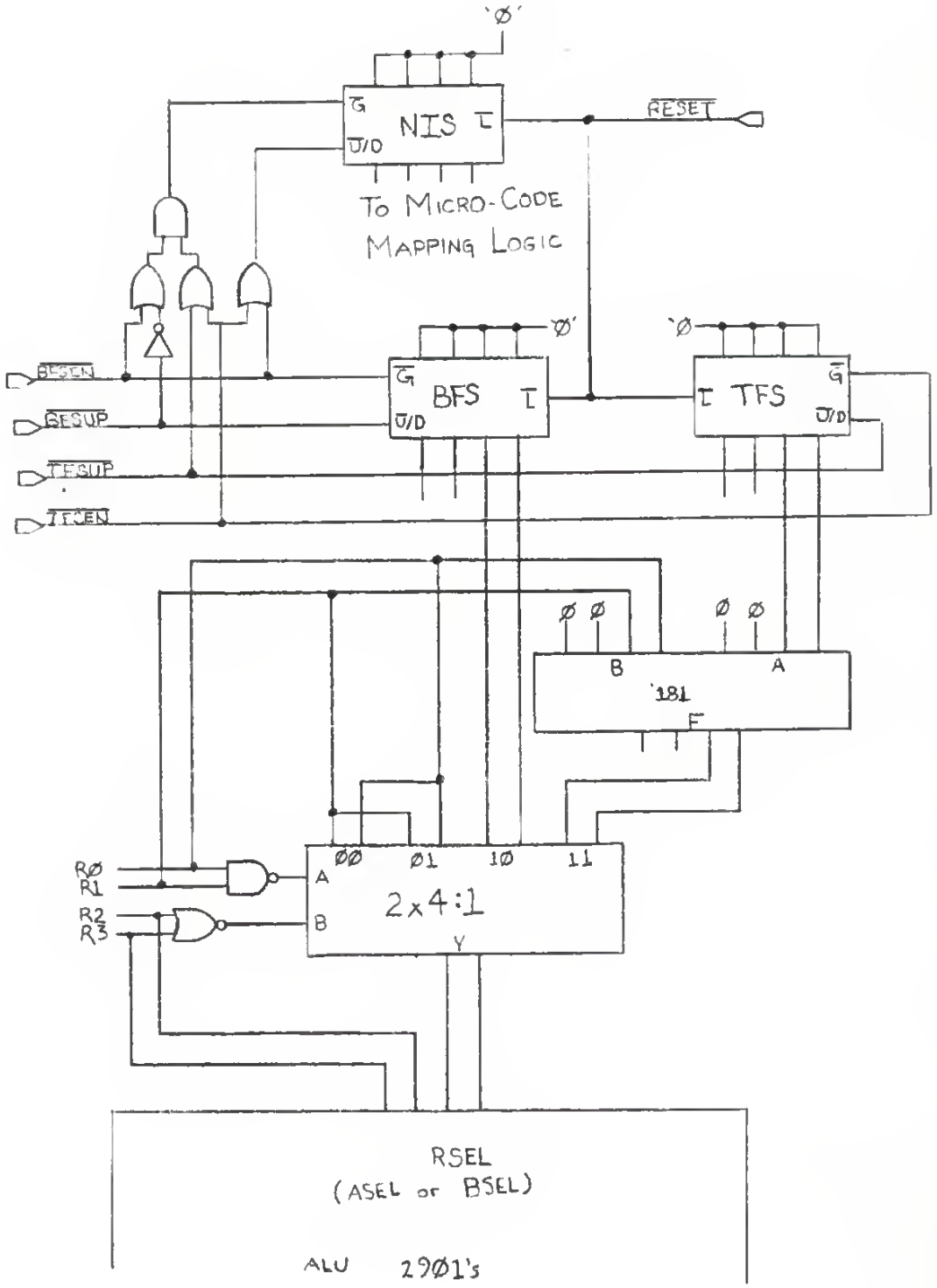


FIGURE 24. FAST STACK ADDRESS MAPPING LOGIC

Register select from PLR	Register select to ALU
0000	(TFS) - 0
0001	(TFS) - 1
0010	(TFS) - 2
0011	(BFS)
0100	0100
0101	0101
0110	0110
0111	0111
1000	1000
1001	1001
1010	1010
1011	1011
1100	1100
1101	1101
1110	1110
1111	1111

Table 6. The Register Mapping Truth Table

The memory stack controller contains the pointers necessary to implement the full stack described in chapter 3 and the logic necessary to perform operations on these pointers. The pointers are: the Top of the Memory Stack (TMS), which points to the first available location in the memory, the current lexical level (LLTOP), the base of the display

registers (DISP), the Base of the Heap (HBASE), and the bottom of the Heap (HBOT). The memory stack controller shares hardware with the ALU instead of being implemented as a separate unit. This adds extra clock cycles for operations which require both an ALU operation and a stack access, but it does eliminate the need for another 32 bit processor and another 20 bits of micro-word. Since the control signals for the ALU already include the capability of addressing any register in the ALU, the ability to perform any operation allowed by the 2901, and the ability to set the carry as needed, there are no extra signals needed to include the function of the memory stack controller in the ALU.

The function of the ALU registers is listed in Table 7.

ALU Register	Function
0000	Part of the Fast Stack
0001	Fast Stack
0010	Fast Stack
0011	Fast Stack
0100	TRUE (TRUE = $FFFFFFFF)
0101	FALSE (FALSE = $00000000)
0110	TWO (TWO = $00000002)
0111	
1000	
1001	TEMP2
1010	TEMP
1011	HBOT
1100	HBASE
1101	LLTOP
1110	DISP
1111	TMS

Table 7. The ALU Register Assignments.

XI. THE INPUT/OUTPUT PROCESSOR

The Input/Output Processor (IOP) has two major functions. It performs all the input and output for the system, and it provides the Console interface. The Console is the device through which the user can control the system. It allows the user to halt the machine and, while the machine is halted, to load data into any of the three memories, to load a value into the pipeline register, the Instruction Register, or the Instruction Counter, or to single step the micro-instruction stream. The Input/Output Unit allows input and output to take place in parallel with the program execution.

The two functions of the IOP are essentially independent. The Console can only be used while the machine is halted, and the I/O unit is only necessary while the machine is running. The two functions both require a processor that is separate from the rest of the system. The Input/Output Unit needs to have the capability of executing a separate instruction stream in order to achieve parallel operation. The Console must be able to execute its routines while the rest of the system is halted. The similar nature and independent functions of the two units allows them to be implemented as a single micro-processor. The two functions will be refered to as the Input/Output Unit (IOU) and the Console Controller (Console) in the discussion. The overall unit will be called the Input/Output Processor.

The IOP is implemented with the Motorola 6802/6846 chip set. The 6802 is an eight-bit micro-processor with a 64K byte address space. The 6846 is a multifunction device, which contains 2K bytes of ROM, an eight bit parallel I/O port, two programmable I/O control lines, and a programmable timer. The ROM "contains" Motorola's MIKBUG monitor. Other parts of the system are four eight-bit I/O ports that interface the IOP to the system bus, eight programmable I/O control lines which serve various functions, RAM which is used to hold the I/O drivers and the Console interface routines, EPROM which holds the modifications to the MIKBUG monitor, and two serial I/O ports for connecting the system to the Console terminal and to some other serial device. A block diagram of the IOP is shown in Figure 25.

The I/O for the machine presented in this paper is defined so that any of 65536 I/O devices might be selected to perform the I/O operation. Each I/O instruction includes a sixteen-bit device code, which will be in either the instruction word immediately following the I/O instruction or on the top of the stack when the I/O instruction is given.

The Input/Output Unit does not use the system clock so data transfers must be coordinated by some form of handshaking. Figure 26 shows the handshake sequence for a write operation and Figure 27 shows it for a read operation. In both operations the system checks the status of the IORDY line. If the IORDY line is true then the IOU is ready and the

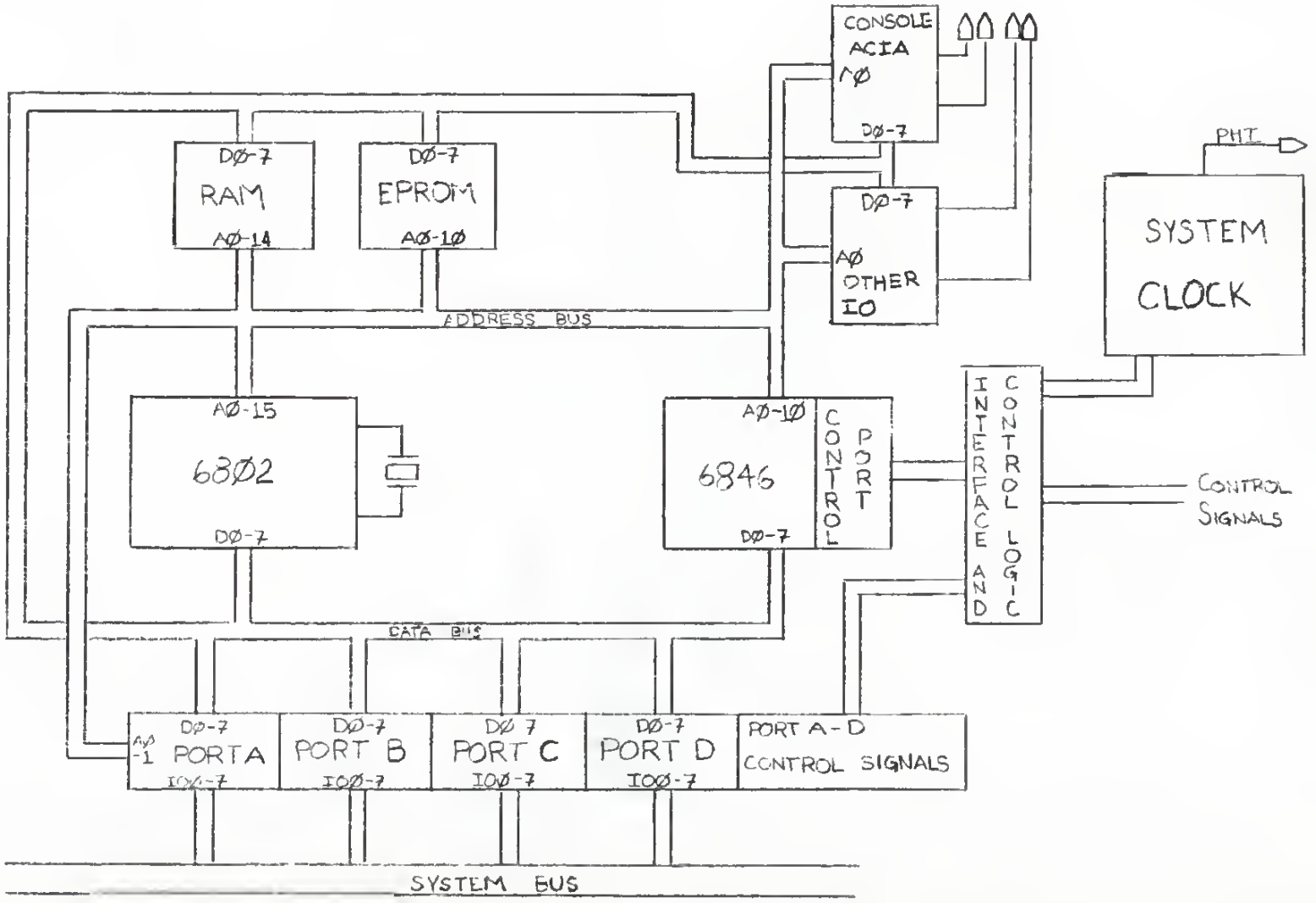


FIGURE 25. THE IOP BLOCK DIAGRAM

operation can continue. If the IOU is not ready, the system goes into a wait loop until the IOU is ready. Once the IOU has signalled that it is ready, the system puts the device code onto the system bus and signals the IOU by raising the DEVSEL signal. DEVSEL is the active high signal that indicates that the data on the bus is a valid device code. The IOU acknowledges the device code by going "not ready", signifying that it has latched the device code. The system can remove the device code once it has seen the IORDY line go low. The DEVSEL must be lowered before or when the device code goes invalid.

The system then waits until the IOU is ready again before continuing. Each transition of a signal by either the IOU or the system must be acknowledged by the other before the handshaking continues. This allows a large difference in clock speeds.

When the IOU has gone ready again, the read operations and write operations become different. For the write operation, the system puts the output data onto the system bus and raises the IODATA line and the WRITE line. This combination of signals indicates that the data on the bus is valid and commands the IOU to read it and output it to the selected device. The IOU signals that it has latched the data by going not ready for a second time. When the system detects the not ready signal, it can remove the data from the bus and lower the IODATA signal. At this point the system is done. The IOU remains not ready until it has finished the Output

operation. For the read operation, the system raises the IODATA line while the WRITE line remains low. This signifies that the system bus belongs to the IOU. The IOU lowers the IORDY line to signify that the data is available on the bus. The system lowers IODATA showing that it has latched the data and that the IOP is free to release the bus. The final step in the read operation is for the system to wait for the bus to be released by the IOU which is indicated by the IORDY going ready again.

Two I/O device codes are reserved. The system needs to know how much memory is available when it is initializing the stack and heap pointers. Input device 0 should always place the bottom-most memory address onto the system bus. Device 1 should always return the top-most memory address.

The IOU must be capable of distinguishing between character data and numeric data. All I/O will be done using characters. The IOU must translate between the numeric values and the character values. In some cases the input or output is to be left in character format. This is indicated by the signal CHAR. CHAR is included in Figures 26 and 27.

The console function requires that the IOP be able to halt the processor. This is done by holding the system clock (PHI) high. The clock circuit is considered a part of the Console since the Console is the only device which can control it. Figure 28 shows the system clock circuit. The clock

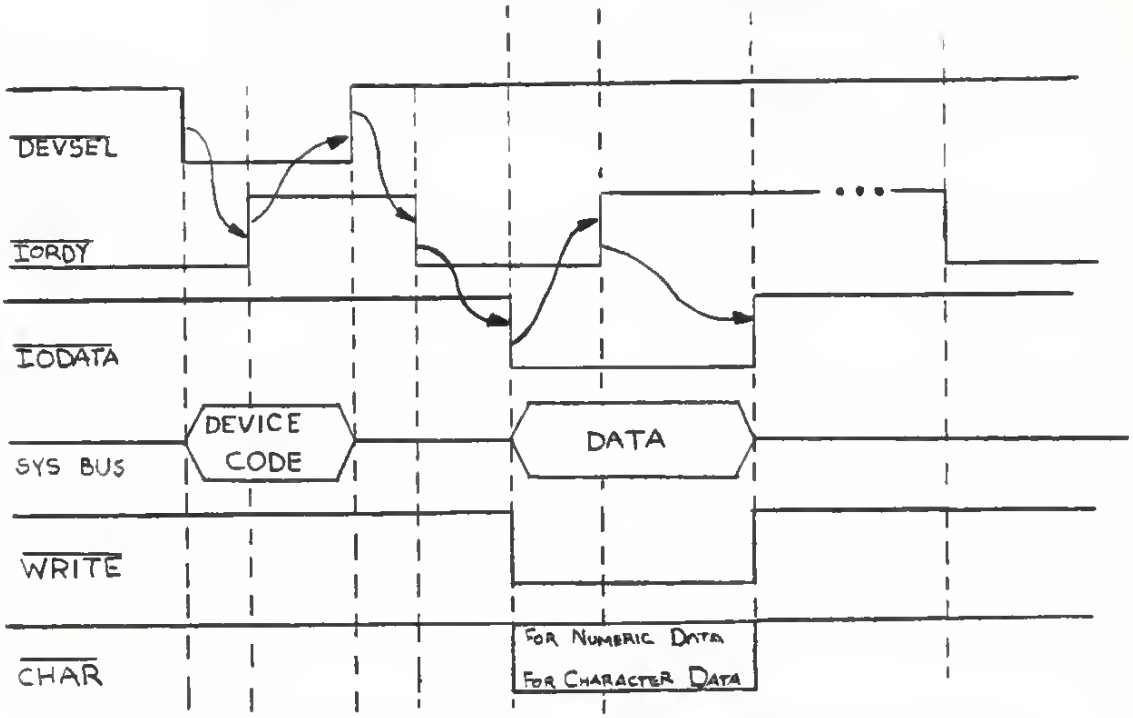


Figure 26. The WRITE Operation Handshake Sequence

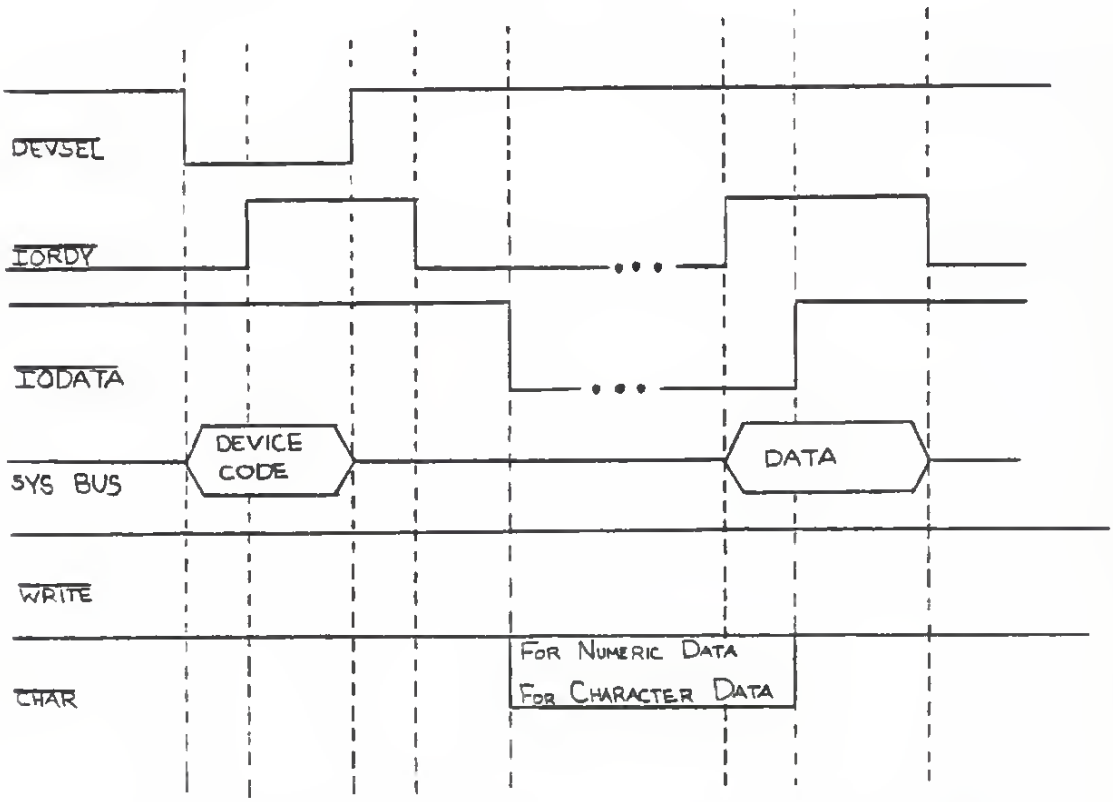


Figure 27. The READ Operation Handshake Sequence

circuit consists of a single JK flip-flop which toggles between 0 and 1, and three other flip-flops which are used to control the clock. The HALTED flip-flop determines whether the clock flip-flop is allowed to toggle. The J input of the clock flip-flop is tied to the inverted output of the flip-flop. The K input is tied to HALTED*. The truth table for the clock flip-flop is:

HALTED*	PHI = Q	Q*	J	K	NEXT
1	0	1	1	1	1
1	1	0	0	1	0
0	0	1	1	0	1
0	1	0	0	0	1

When the machine is running the clock flip-flop toggles back and forth between 1 and 0. When it is halted, the current clock cycle is allowed to complete then the clock is held at 1.

The two remaining flip-flops latch the halt request (HALTRQ*) and single step request (SS*) signals. The HALTRQ flip-flop latches the HALTRQ* signal from the Console. This is latched into HALTRQ on a rising edge on the PHIC line from the console. The output of the HALTRQ flip-flop passes through the AND gate and is latched into the HALTED flip-flop on the next rising edge of the oscillator. The clock is then halted as soon as the current clock cycle finishes.

The Console must provide the clock when the system is

halted. Many of the latches in the machine are triggered on the rising edge of the system clock. In order for the Console to load the Instruction Register, for example, it must supply the clock edge that causes the IR to latch the data at its inputs. The Console supplies this clock by raising and lowering the signal PHIC. PHIC replaces PHI when the system is halted. PHIC is multiplexed onto the PHI line by the OR and AND gates in Figure 28.

The single-step request flip-flop (SS) is set asynchronously by the STEP* signal from the Console. This causes the HALTED flip-flop to be reset by the next rising edge of the oscillator. The same rising edge also resets the SS flip-flop so that only one clock cycle is allowed before the system is halted again.

All the flip-flops in the clock and clock control circuits are either set or reset by the RESET signal. This allows the system to start up in the HALTED state, with the clock in a predictable state.

When the system has been halted, the Console takes over the function of many of the system control signals. Table 8 shows the sytem signals that are controlled by the console when the machine is halted. There are also a number of functons that the Console can perform that are not available when the machine is running. Table 9 lists the signals that perform these functions. All the functions in Tables 8 and 9 are independent. They all put or get their data from the system

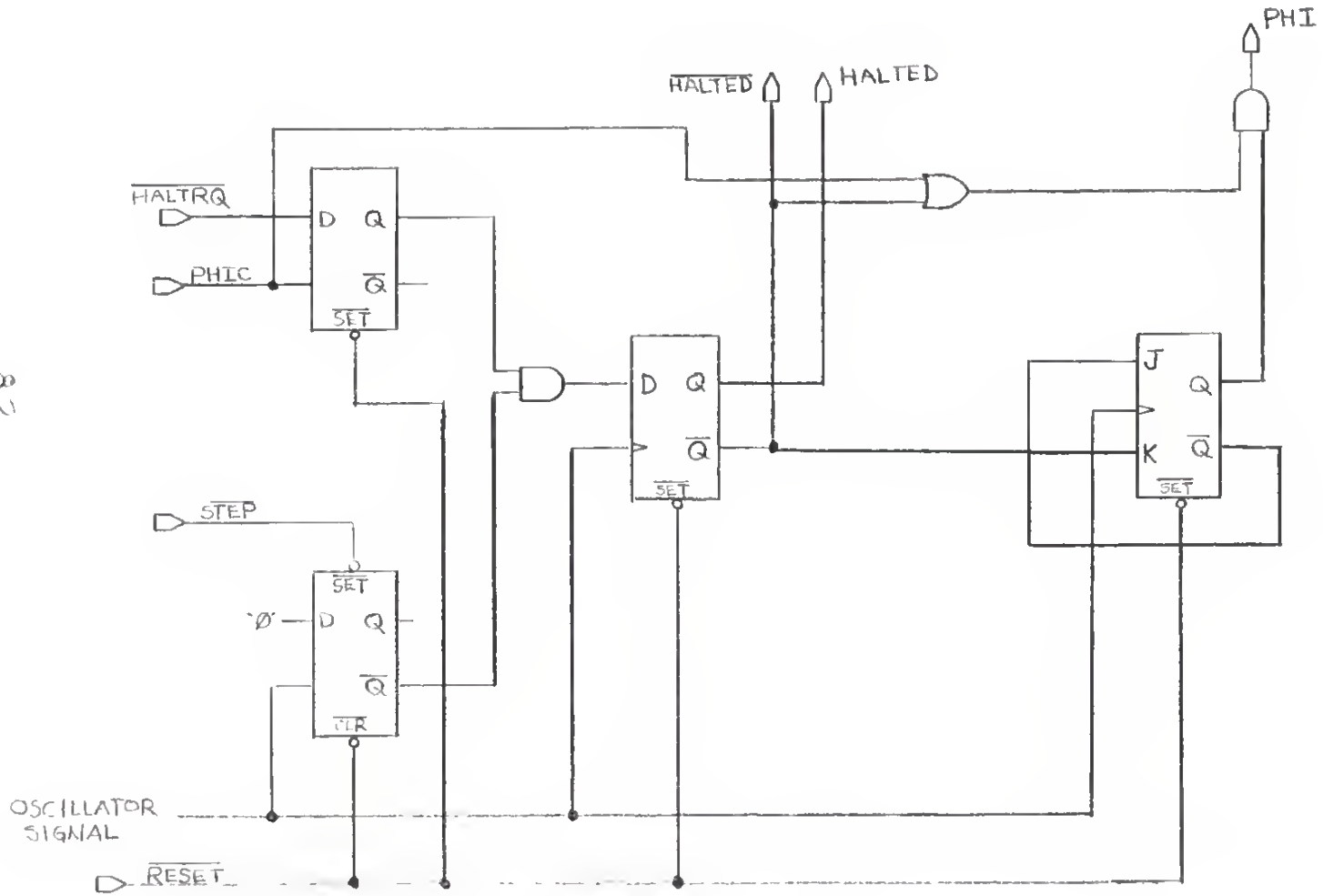


FIGURE 28. THE SYSTEM CLOCK CIRCUIT

Signal	Function
MEMSEL*	Enable the Data Memory for reading or writing.
ENMAR*	Enable the Memory Address Register to be loaded on the next rising edge of PHI.
ENIR*	Enable the Instruction Register to be loaded on the next rising edge of PHI.
SAVEIC*	Enable the Instruction Counter on to the system bus.
LOADIC*	Load a value into the Instruction Counter. (includes JUMP and FORCE from the PLR)

Table 8. Control Signals Taken Over By the Console

Signal	Function
ENPLR0*	Enable the low half of the PLR to be loaded on the next rising edge of PHI.
ENPLR1*	Enable the high half of the PLR to be loaded on the next rising edge of PHI.
ENCSAR*	Enable the Control Store Address Register to be loaded on the next rising edge of PHI.
ENIMAR*	Enable the Instruction Memory Address Register to be loaded on the next rising edge of PHI.

Table 9. Control Signals Generated By the Console

bus. Only one should be performed at a time. The signals can come from a decoder. Figure 29 shows the generation of all the signals IOP signals. The 4-to-16 decoder is enabled by the HALTED* signal, so if the system is not halted, all of its signals will be high.

The signals in Table 8 must be combined with the signals from the PLR to generate the system signals. Figure 30 shows how this is done for each of the signals. All of the PLR signals are active high and all the Console signals are active low. The system signals are active low except for the JUMP and FORCE signals.

The signals in Table 9 can be used directly. There are no PLR signals for them to interface with. However, the two ENPLR* signals must be low when the machine is not halted, so they are ANDed with HALTED.

Both the IR and the PLR have two possible sources of inputs when they are loaded under control of the console. The data can come from the system bus or from the respective instruction store. The bus transceivers need to be enabled if the data is to come from the system bus. Therefore, the signal to load the instruction register and the signal to enable the bus transceiver are not independent and can not both come from the decoder. The signals IMR/W and the pair CSR/W0 and CSR/W1, which come directly from the control port and not the decoder, are used to enable the bus transceivers for the respective instruction memories.

There are two other signals that work in conjunction with the others. WRITE determines in which direction the bus transceivers are enabled, and provides the write enable signal to all the memories. PHIC provides the clock signal as described earlier.

The eight-bit parallel port in the 6846 is used as the control port. Four of the bits are used as inputs to the decoder. The other four bits are used directly as the IMR/W, CSR/W0, CSR/W1, and HALTRQ signals. The eight-bit I/O port in the 6846 and each of the four eight-bit ports used to interface the IOP to the system bus have two control lines with them. One these control lines for each port is always input and the other is programmable to either input or output. These control lines are used to provide some of the control signals and to receive the control signals to the IOP.

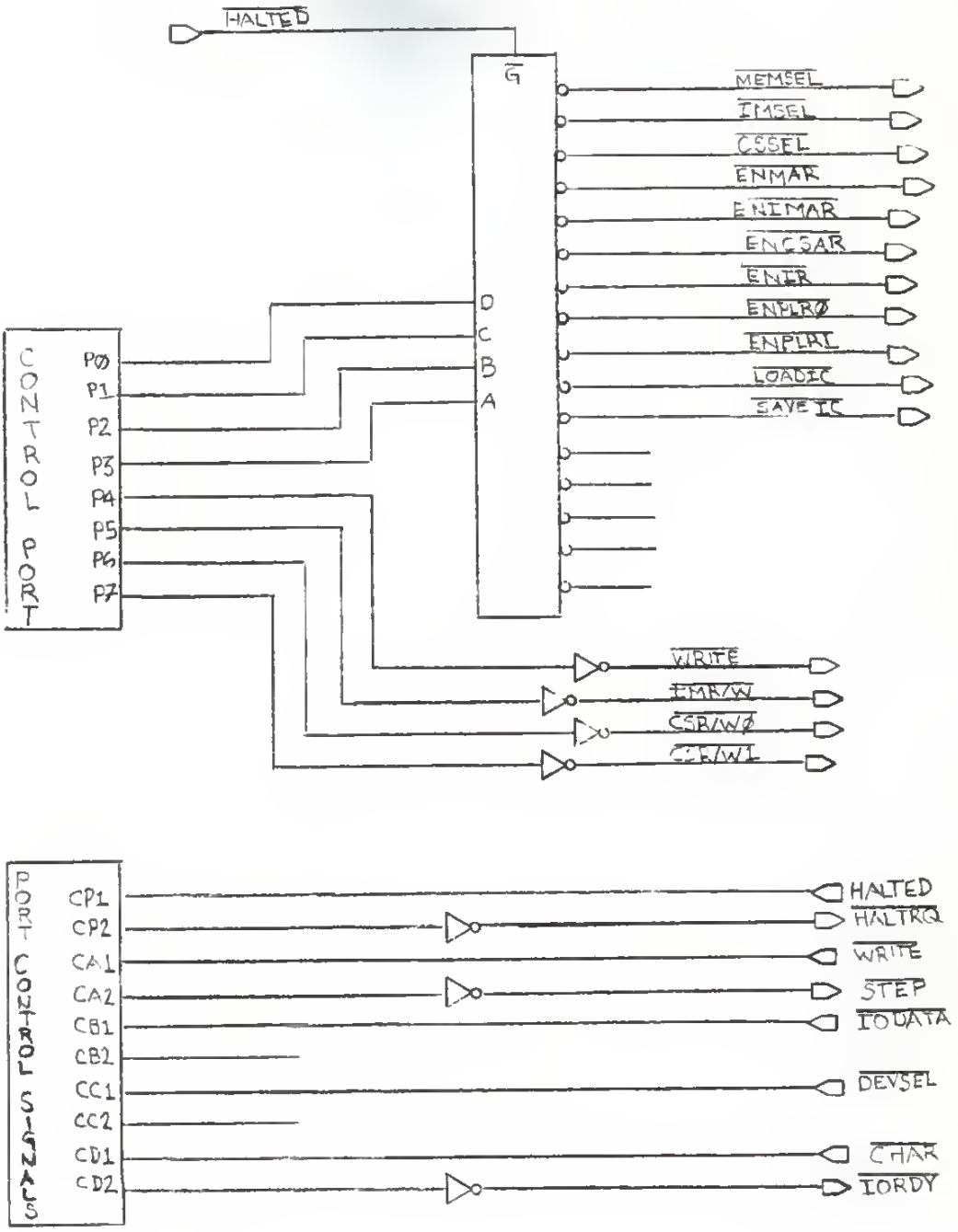


FIGURE 29. THE IOP CONTROL SIGNALS

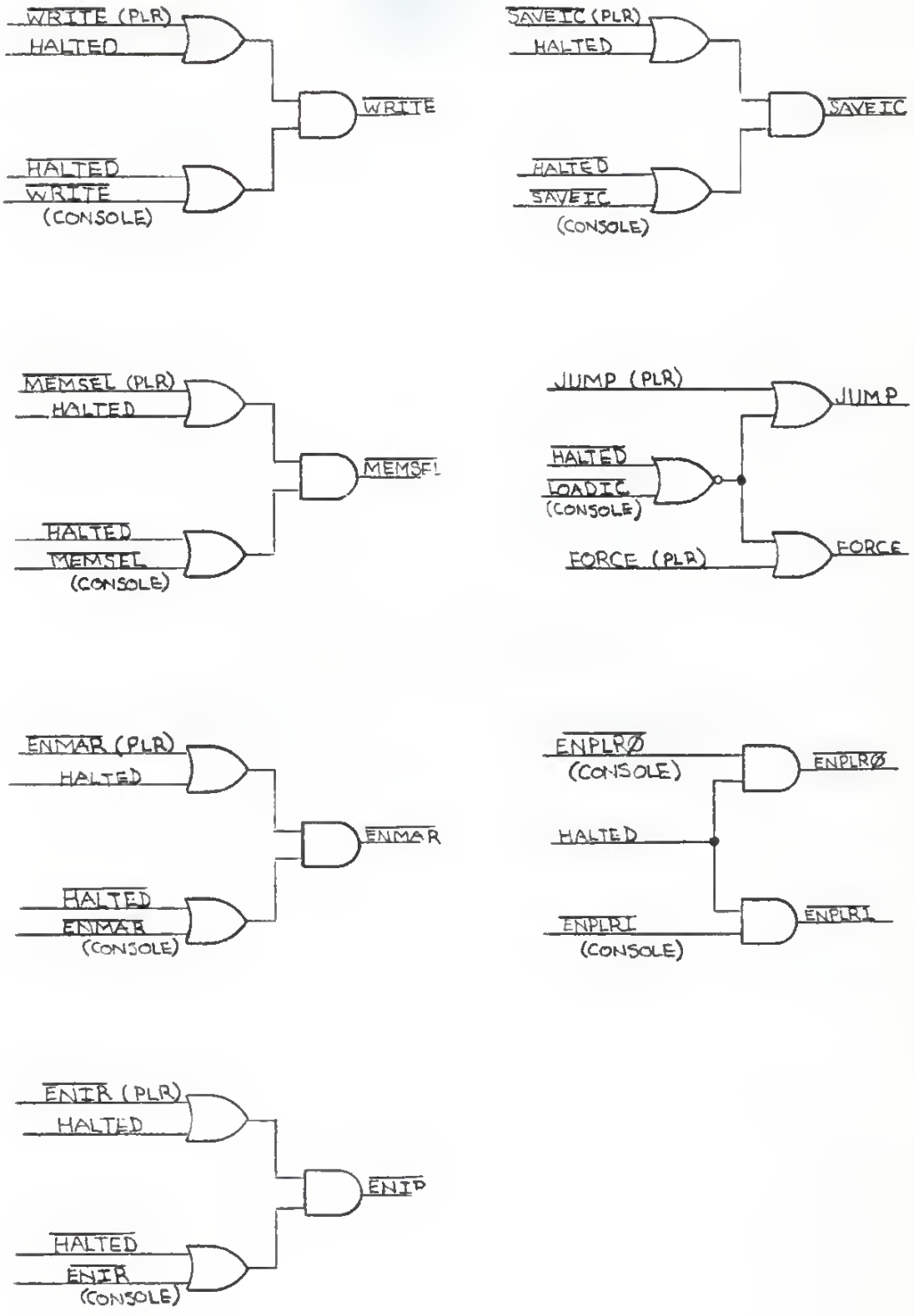


FIGURE 30. INTERFACING THE CONSOLE SIGNALS

XII. THE MICRO INSTRUCTIONS.

The micro instructions for this machine are 64 bits wide. Instead of giving the bit coding for each instruction, a language is defined and then the micro-instructions are given using this language. A translation from the language to the bit patterns is also given.

Each micro-instruction has the general format:

```
NNN   TITLEn   ALU   : A=        Cn=        F=        B
                       B=        D=         Y=        Q
               STACK:
               M     :
               BUS   :
               uPU   :
               IPU   :
               IOP   :
                    English explanation
```

Every micro-instruction has a name and an address in the control store associated with it. NNN is the control store address. It is a three digit hex number that is unique for each micro-instruction. TITLEn is the name of the micro-instruction. TITLE is the same as the mnemonic for the macro-instruction which is being implemented. n is the number of the instruction in the sequence. Thus, the second micro-instruction in the sequence that implements the CALL PROCEDURE instruction (mnemonic is CALL) is CALL2.

The ALU, STACK, M, uPU, IPU, and IOP fields correspond to fields in the Pipeline Register. The BUS field only directly specifies one signal which is best included with the IPU control signals. For all the fields an X indicates "Do

Nothing". For many cases, Do Nothing is the same thing as "Don't Care". However the ALU and the IOP require specific siganls to "Do Nothing". The signals that should be given for a Do Nothing state are given in the discussion of the individual fields.

In the machine that is being microprogrammed, some operations occur at the end of the clock cycle. Most registers are latched at this point. Some signals are required for the entire clock cycle. To distinguish between events and signals, the notation is that the symbol "<-" is used to indicate that the operation occurs at the end of the clock cycle, and "=" is used to indicate that the signal should maintain its level throughout the clock cycle. For example, the IPU field often looks like:

IPU: IR = IR1 ; IC <- IC+1 ; IR <- IM(IC)

The IR = IR1 field indicates that during the clock cycle, anything that references the IR will see the IR1 version of it. IC <- IC+1 indicates that the Instruction Counter is to be incremented at the end of the clock cycle. If the IC contained 198 as the instruction was beginning, the value loaded into the IR by the IR <- IM(IC) would be the value fetched from location 198, because the IC is not incremented until the end of the cycle.

The ALU field of the PLR contains the control signals for both the ALU and Memory Stack Controller, since these two

functions share hardware. Therefore, the ALU field in the language used to describe the micro-instructions includes both the ALU and the Memory stack functions.

The A field is the A operand selection for the 2901's. There are fourteen possible selections for the A operand. The selection from the PLR is mapped by the fast stack control logic so that the fast stack elements can always be selected relative to either the top or the bottom of the fast stack. The micro-instruction does not need to know how many words are in the stack nor where the top or bottom is in the ALU registers.

The format for the translation has the field from the descriptive language named on the right and the corresponding PLR field named on the left. The possible choices are given below the language field, and the bit pattern is given below the PLR field name.

The options for the A field are:

A	ASEL
TOP	0000
TOP-1	0001
TOP-2	0010
BOTTOM	0011
TRUE	0100
FALSE	0101
TWO	0110
TEMP2	1001
TEMP	1010
HBOT	1011
HBASE	1100
LLTOP	1101
DISP	1110
TMS	1111
X	Don't Care (typically 0000)

The B field specifies the B operand selction for the ALU 2901's. The translation from B to BSEL in the PLR is the same as the A to ASEL translation.

The Cn field specifies how the carry into the least significant slice is to be set. It only has an effect on the add and subtract operations in the 2901's. The Cn translation is:

Cn	Cn
0	0
1	1
X	Don't Care (typically 0)

The D field selects which of the two possible sources is routed through the 2-to-1 multiplexers to the direct data inputs (D inputs) of the ALU 2901's.

D	DSEL
BUS	0
IR	1
X	Don't Care (typically 0)

The F field selects the operation that is performed by the 2901's. The 2901 literature defines two separate fields for the operand selection and the function selection. These are combined here.

F	I5-I0	F	I5-I0
A+B	000001	AVB	011001
Q+0	000010	QV0	011010
B+0	000011	BV0	011011
A+0	000100	AV0	011100
D+A	000101	DV0	011111
B-A	001001	B&A	100001
Q-0	001010	Q&0	100010
A-0	001100	B&0	100011

A-D	001101	A&0	100100
0-B	010011	B*	111011
0-A	010100	X	Don't Care (typically 000000)

The add and subtract operations also include the carry. They are actually R+S+Cn and R-S-Cn* or S-R-Cn*.

The Y,Q, and B fields together define the 2901 destination select field and the 2901 output field. The destination selection field has the capability of shifting either left or right before loading into either the register selected by the BSEL field or into the Q register. The Y outputs of the 2901 can be either the result of the operation or contents of the register selected by the ASEL field. However, the latter option is constrained so that there must also be a load into the register selected by the BSEL field and there can be no shifts. The translation is:

Y	ALUOE	
F or A	1	(The output is enabled.)
None	0	(The output is disabled.)

Y	B	Q	I8-I6
None or F	No	Q <- F	000
None or F	No	No	001
A	B <- F	No	010
None or F	B <- F	No	011
None or F	B <- F/2	Q <- Q/2	100
None or F	B <- 2F	No	111

The final field in the ALU field is called MULT. It enables the external logic that simplifies the multiplication operation. If MULT appears in the ALU field, then MULT = 1. Otherwise MULT = 0.

The Do Nothing state for the ALU is:

```
ASEL = X
BSEL = X
Cn = X
DSEL = X
I2-I0 = X
I5-I3 = X
I8-I6 = 001
MULT = X
ALUOE = 0
```

The stack field defines what operations are performed on the fast stack pointers. The fast stack is specified by three registers. The BFS pointer points to the register that contains the bottom of the fast stack (Oldest element). The TFS pointer points to the register that will receive the next element at the Top of the Fast Stack. And the NFS counts the Number of elements in the Fast Stack. The translation is:

STACK	TFSEN	TU*/D	BFSEN	BU*/D	NFS
TFS <- TFS+1	1	0	0	X	UP
TFS <- TFS-1	1	1	0	X	DOWN
BFS <- BFS+1	0	X	1	0	DOWN
BFS <- BDS-1	0	X	1	1	UP
X (Do Nothing)	0	X	0	X	NONE

(X = Don't Care, typically 0)

The M field specifies the control signals for the memory. The translation is:

M	MEMSEL	ENMAR	WRITE
LOAD MAR	0	1	X
READ	1	0	0
WRITE	1	0	1
X	0	0	X

The BUS field, as mentioned earlier, only actually controls one of the signals specified in it. The rest of the

signals are just informational. When the BUS field is IR, then the IROE signal is 1. The meanings of the BUS signals are:

ALU	The ALU output is put on the bus.
IR	The Instruction Register is enabled onto the bus.
M	The output of the memory is put on the bus.
IC	The output of the instruction sequencer is put on the bus.
IOP	The IOP has put the data on the bus.
X	The bus is carrying no data.

The micro-processing unit allows 16 different commands, of which only ten are used. The Micro-sequencing unit commands are given using the standard 2910 mnemonics. The mnemonics are expanded below the translation table. Many of the commands use an external input as either a branch address or a counter value. For all the instructions used, except the JMAP instruction, this external input comes from the Pipeline register. When the value is to be used as a branch address, the name of the destination instruction is given instead of the address. This makes the code a little easier to read. When the value is to be loaded into the counter, it is given in hex. In either case the field is given as PL = followed by the appropriate label or value. The only other field of micro-processing unit control is the condition code select field. This field is used to route one of sixteen possible signals to the CC* input of the 2910. The field is specified by COND = followed by the condition that is to be tested.

OPERATION	I3-I0
CJS	0001
JMAP	0010
CJP	0011
PUSH	0100
JRP	0111
RFCT	1000
RPCT	1001
CRTN	1010
LDCT	1100
CONT	1110

CJS = Conditional Jump to Subroutine
JMAP = Jump to an address from the Mapping logic
CJP = Conditional Jump to an address from the PLR
PUSH = PUSH the next address onto the stack and conditionally load the counter register
JRP = Conditional Jump to an address from the Register or the PLR.
RFCT = Repeat loop branching to an address from the File until the counter reaches 0
RPCT = Repeat loop, branching to an address from the PLR, intil the counter reaches 0
CRTN = Conditional Return from subroutine
LDCT = Load Counter
CONT = Continue

COND =	CCSEL	
equal	0000	
negative	0001	
positive	0010	
not positive	0011	
not negative	0100	
not equal	0101	
EMPTY*	0110	
IORDY	0111	
IORDY*	1000	
forced	1111	
NOT SHOWN	XXXX	(typically 0000)

The instruction processor requires several control signals to perform a simple task. The signal names by themselves do not give a very clear picture of what is going on. The language used to describe the operations tries to indicate the event more than the signals that cause the event.

Thus, the translation is not as simple as the case of the condition signals, for example. The two PLR signals NEXT and JUMP determine whether the next instruction is fetched from the address following the address of the current instruction or whether the address of the next instruction is to be loaded from somewhere. FORCE is a PLR signal that forces the jump to be taken. The CONDITIONAL BRANCH only takes the jump if the value on the top of the stack is FALSE. All other jumps are forced. The IPU operation translations are:

OPERATION	NEXTI	JUMP	FORCE
IC <- IC+1	1	0	0
JUMP	0	1	0
JUMP forced	0	1	1

Another field in the IPU section specifies which of the Instruction Registers is to be used. The field appears as "IR = IRn". This translates as:

IR	IRSEL
IR1	0
IR2	1
X	Don't Care (typically 0)

The low order eight bits of IR2 can be used as the lexical level of a variable. The upper eight bits are masked off by raising the PLR signal MASK. This is indicated by "MASKED(IR2)" in the IPU field.

Other operations are the loading of the Instruction Register which is indicated by "IR <- IM(IC)" in the IPU field and results in ENIR = 1 in the microcode, and SAVEIC which is indicated by "SAVEIC" in the IPU field and causes SAVEIC = 1 in

the PLR.

The Do Nothing state is:

NEXTI	0
JUMP	0
FORCE	0
SAVEIC	0
ENIR	0
IROE	0
MASK	X
IRSEL	X

The Input/Output Processor control field handles the handshaking between the system and the IOP. The WRITE signal is the same signal as is used by the memories, and is a 1 for an output operation and a 0 for an input operation. These are indicated by "WRITE" and "READ" respectively. For the other signals, if the signal is named in the IOP field then it is to be a 1 in the PLR. The Do Nothing state is with all signals at 0.

The English explanation is optional. It is included when there is some new information available. It is omitted if a certain operation requires more than one step but has been described in the first.

The micro-code routines follow:

The initilalization routine is located at location zero to simplify the console's task of starting a program. It can be done by loading the PLR such that I0-I3 for the uPU are all zeros.

```
000    INIT1     ALU   : X
                 STACK: X
                 M     : X
                 BUS   : X
                 uPU   : CJP  ;  PL = INIT2  ;  COND = forced
                 IPU   : X
                 IOP   : X
                         Get out of the way while it is convienent
                         Continued at location 007.
```

The following routines do not require any operands or any empty locations in the fast stack.

```
002    JUMP1     ALU   : X
                 STACK: X
                 M     : X
                 BUS   : IR
                 uPU   : CJP  ;  PL = DLY1  ;  COND = forced
                 IPU   : IR = IR1  ; JUMP  ; IR <- IM(IC)
                 IOP   : X
                         Load IC then wait to allow decode

004    IHEAP1    ALU   : A= HBOT   Cn= 0     F= D+A    B <- F
                         B= HBOT   D=  IR    Y= None   Q No
                 STACK: X
                 M     : X
                 BUS   : X
                 uPU   : CJP  ;  PL = DLY1  ;  COND = forced
                 IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                 IOP   : X
                         Incrementthen wait for n to get outof
                         the way.

006    DHEAP1    ALU   : A= HBOT   Cn= 1     F= A-D    B <- F
                         B= HBOT   D=  IR    Y= None   Q No
                 STACK: X
                 M     : X
                 BUS   : X
                 uPU   : CJP  ;  PL = DLY1  ;  COND = forced
                 IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                 IOP   : X
                         Decrement and wait for fetch.
```

The following is the remainder of the initialization routine.

```
007   INIT2    ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = INIT2 ;  COND = IORDY
               IPU   : X
               IOP   : X
                       Get bottom of memory from IOP.

008   INIT3    ALU   : A= FALSE  Cn= 0     F= A&0    B <- F
                       B= FALSE  D=  X     Y= F      Q No
               STACK: X
               M     : X
               BUS   : ALU
               uPU   : CJP  ;  PL = INIT3 ;  COND = IORDY
               IPU   : X
               IOP   : DEVSEL
                       Device 0 always returns bottom of memory.

009   INIT4    ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = INIT4 ;  COND = IORDY*
               IPU   : X
               IOP   : X

00A   INIT5    ALU   : X
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CJP  ;  PL = INIT5 ;  COND = IORDY
               IPU   : X
               IOP   : IODATA  ;  READ

00B   INIT6    ALU   : A= X      Cn= X     F= DV0    B <- F
                       B= DISP   D=  BUS   Y= None   Q No
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CONT
               IPU   : X
               IOP   : IODATA  ;  READ
                       Read in the bottom of memory.

00C   INIT7    ALU   : X
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CJP  ;  PL = INIT7 ;  COND = IORDY*
               IPU   : X
               IOP   : X
                       Wait for the IOP to release the Bus.
```

```
00D   INIT8    ALU   : A= FALSE  Cn= 1     F= A+0    B <- F
                       B= TWO    D=  X     Y= F      Q No
               STACK: X
               M     : X
               BUS   : ALU
               uPU   : CJP  ;  PL = INIT8 ;  COND = IORDY
               IPU   : X
               IOP   : DEVSEL
                       Get the top of memory from device 1.

00E   INIT9    ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = INIT9 ;  COND = IORDY*
               IPU   : X
               IOP   : X

00F   INIT10   ALU   : X
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CJP  ;  PL = INIT10  ;  COND = IORDY
               IPU   : X
               IOP   : IODATA  ;  READ

010   INIT11   ALU   : A= X      Cn= X     F= DV0    B <- F
                       B= HBASE  D=  BUS   Y= None   Q No
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CONT
               IPU   : X
               IOP   : IODATA  ;  READ
                       Read in the top of memory.

011   INIT12   ALU   : X
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CJP  ;  PL = INIT12  ;  COND = IORDY*
               IPU   : X
               IOP   : X
                       Wait for IOP to release the Bus.

012   INTI13   ALU   : A= TWO    Cn= 1     F= A+0    B <- F
                       B= TWO    D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Load the constant 2 into TWO.
```

```
013   INIT14   ALU   : A= FALSE  Cn= 0     F= 0-A    B <- F
                       B= TRUE   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Load the constant TRUE into TRUE.

014   INTI15   ALU   : A= FALSE  Cn= 0     F= A-0    B <- F
                       B= LLTOP  D=  X     Y= A      Q No
               STACK: X
               M     : X
               BUS   : ALU
               uPU   : CONT
               IPU   : JUMP  ;  Forced
               IOP   : X
                       Set the current lexical level to -1.
                       Set the IC to 0.

015   INIT16   ALU   : A= TWO    Cn= X     F= AV0    B <- 2F
                       B= TEMP   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : PUSH  ;  PL = 5  ;  COND = forced
               IPU   : X
               IOP   : X

016   INIT17   ALU   : A= X      Cn= X     F= BV0    B <- 2F
                       B= TEMP   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : RFCT  ;  PL = INIT17
               IPU   : X
               IOP   : X
                       Calculate the constant 256 in TEMP.

017   INIT18   ALU   : A= DISP   Cn= 0     F= A+B    B <- F
                       B= TEMP   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Calculate the bottom of the stack.
```

```
018   INIT19   ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                       B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Initialize the Top of memory stack
                       pointer to the bottom of the stack.
                       Load first MI into IR.

019   INIT20   ALU   : A= HBASE  Cn= X     F= AV0    B <- F
                       B= HBOT   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Go begin executing.
```

The following are the fix routines that maintain the fast stack so that the microcode routines that perform a macro-instruction always get the operands in the fast stack that they need or the empty location that they need. The mapping logic compares the number of elements in the fast stack to the number of elements needed for the macro-instruction. If there are not enough or too many, the mapping logic forces the address of the appropriate fix routine onto the external inputs of the micro-sequencer.

FIXIN is the routine that brings in an operand when there are not enough operands in the fast stack for an operation to continue.

```
01A   FIXIN1   ALU   : A= TMS    Cn= 0     F= A-0    B <- F
                       B= TMS    D=  X     Y= F      Q No
               STACK: BFS <- BFS-1
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Fetch the top memory stack element.
```

```
01B   FIXIN2   ALU   : A= X       Cn= X      F= DV0     B <- F
                      B= BOTTOM D=  BUS    Y= None    Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : JMAP
               IPU   : IR = IR2
               IOP   : X
                       And load it into the bottom of the fast
                       stack.
```

FIXOUT is the routine that writes out the bottom element of the fast stack when there is not a vacant location to write the next operand into.

```
01C   FIXOUT1  ALU   : A= TMS     Cn= 1      F= A+0     B <- F
                      B= TMS     D=  X      Y= A       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Write out the bottom element of the
                       fast stack.

01D   FIXOUT2  ALU   : A= BOTTOM Cn= X      F= AV0     B <- F
                      B= BOTTOM D=  X      Y= A       Q No
               STACK: BFS <- BFS+1
               M     : WRITE
               BUS   : ALU
               uPU   : JMAP
               IPU   : IR = IR2
               IOP   : X
```

FLUSH is the routine that empties the fast stack. It is invoked by the mapping logic when the macro-instruction is one which causes a change in context or one that moves the top of stack pointer. If the Top of stack pointer changes for either reason, then the values in the fast stack must be written out to the memory stack before the pointer is changed or they will end up being written to the wrong location.

```
01E   FLUSH1   ALU   : A= TMS     Cn= 1      F= A+0     B <- F
                      B= TMS     D=  X      Y= A       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
```

```
01F   FLUSH2   ALU   : A= BOTTOM Cn= X      F= AV0     B <- F
                      B= BOTTOM D=  X      Y= A       Q No
               STACK: BFS <- BFS+1
               M     : WRITE
               BUS   : ALU
               uPU   : CJP  ;  PL = FLUSH1  ;  COND = EMPTY*
               IPU   : X
               IOP   : X
                       Write out the bottom element until the
                       stack is empty.

020   FLUSH3   ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR2
               IOP   : X
```

The following are the work subroutines that are called from various locations in the microcode.

The following routine is used to allow the instruction mapping logic time to decode the macro-instruction. If the instruction register is latched at the end of the clock cycle, then another clock cycle must be allowed before the micro-instruction routine to perform the macro-instruction can be called.

```
021   DLY1     ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
```

ADDEV is the routine to evaluate the two part variable address. The routine requires that the address of the display register be in the MAR and the variable offset, j, be in IR2. The routine returns with the address of the variable in the MAR.

```
022   ADDEV1   ALU   : A= X      Cn= X     F= DV0     B <- F
                      B= TEMP   D=  BUS   Y= None    Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Read the contents of the display
                       register.
```

```
023   ADDEV2   ALU   : A= TEMP   Cn= 0     F= D+A    B No
                       B= X      D=  IR    Y= F      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CRTN  ;  COND = forced
               IPU   : IR = IR2
               IOP   : X
                       Add the variable displacement to the
                       display register contents and store the
                       result into the MAR.
```

The following routine will set the display registers according to the static display chain, given the address of a transfer point in the Q register.

```
024   SCOPE1   ALU   : A= X      Cn= 0     F= Q-0    B NO
                       B= X      D=  X     Y= F      Q <- F
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Fetch the return address.

025   SCOPE2   ALU   : X
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CONT
               IPU   : LOAD  ;  forced
               IOP   : X
                       Load the return address into the IC.

026   SCOPE3   ALU   : A= X      Cn= 0     F= Q-0    B NO
                       B= X      D=  X     Y= F      Q <- F
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Fetch the lexical level.  Fetch the
                       instruction at the return address.
```

```
027   SCOPE4   ALU   : A= X       Cn= X     F= DV0     B <- F
                       B= LLTOP   D=  BUS   Y= None    Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Load the lexical level from the stack
                       into the current lexical level register.

028   SCOPE5   ALU   : A= X       Cn= 0     F= Q-0     B NO
                       B= X       D=  X     Y= F       Q <- F
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Fetch the value that is to be restored
                       into the current display register.

029   SCOPE6   ALU   : A= X       Cn= X     F= DV0     B <- F
                       B= TEMP2   D=  BUS   Y= None    Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Save the display register value in a
                       temporary location that makes the
                       loop more efficient.

02A   SCOPE7   ALU   : A= DISP    Cn= 0     F= A+B     B NO
                       B= LLTOP   D=  X     Y= F       Q <- F
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       No longer need stack pointer for this
                       routine.  Use the Q register as another
                       temporary that contains a pointer to the
                       display register that is being restored.
```

```
02B   SCOPE8    ALU   : A= TEMP2  Cn= X     F= AV0    B <- F
                      B= TEMP2  D=  X     Y= A      Q No
                STACK: X
                M     : WRITE
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X
                        Restore the display register value into
                        the display register.

02C   SCOPE9    ALU   : A= LLTOP  Cn= X     F= AV0    B <- F
                      B= TEMP   D=  X     Y= None   Q No
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = SCOPE14  ;  COND = equal
                IPU   : X
                IOP   : X
                        Set up the WHILE (TEMP NOT= 0) loop.

02D   SCOPE10   ALU   : A= X      Cn= 0     F= B-0    B <- F
                      B= TEMP2  D=  X     Y= F      Q No
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X
                        The value to be restored into the next
                        lower display register is stored one
                        stack location below where the current
                        or just restored display register points.

02E   SCOPE11   ALU   : A= X      Cn= X     F= DV0    B <- F
                      B= TEMP2  D=  BUS   Y= None   Q No
                STACK: X
                M     : READ
                BUS   : M
                uPU   : CONT
                IPU   : X
                IOP   : X
                        Get the value to be restored.

02F   SCOPE12   ALU   : A= X      Cn= 0     F= Q-0    B NO
                      B= X      D=  X     Y= F      Q <- F
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X
                        Put the address of the next lower display
                        register into the MAR and back into Q.
```

```
030   SCOPE13  ALU   : A= TEMP2  Cn= X     F= AV0    B <- F
                       B= TEMP2  D=  X     Y= A      Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Restore the value saved a little bit ago
                       into the display register.

031   SCOPE14  ALU   : A= X      Cn= 0     F= B-0    B <- F
                       B= TEMP   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = SCOPE10 ;  COND = not equal
               IPU   : X
               IOP   : X
                       TEMP is the loop index.  Decrement it
                       and see if the display register
                       corresponding to lexical level 0 has
                       been reached.

032   SCOPE15  ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
```

The following routines all require that at least one operand be in the fast stack.

```
040   NEG1     ALU   : A= X      Cn= 1     F= 0-B    B <- F
                       B= TOP-1  D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ; IR <- IM(IC)
               IOP   : X
```

```
042   NOT1      ALU   : A= X       Cn= 0      F= (0XORB)*  B <- F
                      B= TOP-1    D=  X      Y= NONE      Q No
                STACK: X
                M     : X
                BUS   : X
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X

044   LOAD1     ALU   : A= TOP-1   Cn= X      F= AV0     B <- F
                      B= TOP-1    D=  X      Y= A       Q No
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X

045   LOAD2     ALU   : A= X       Cn= X      F= DV0     B <- F
                      B= TOP-1    D=  BUS    Y= NONE    Q No
                STACK: X
                M     : READ
                BUS   : M
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <-IM(IC)
                IOP   : X

046   COND1     ALU   : A= TOP-1   Cn= X      F= AV0     B NO
                      B= X        D=  X      Y= NONE    Q No
                STACK: TFS <- TFS-1
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = DLY1  ;  COND = forced
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X

048   SDATA1    ALU   : A= DISP    Cn= 0      F= D+A     B NO
                      B= X        D=  IR     Y= F       Q NO
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CJS  ;  PL = ADDEV1  ;  COND = forced
                IPU   : IR = MASKED(IR2) ; IC <- IC+1 ;
                           IR <- IM(IC)
                IOP   : X
```

```
049   SDATA2   ALU   : A= TOP-1  Cn= X     F= AV0    B <- F
                      B= TOP-1  D=  X     Y= None   Q No
               STACK: TFS <- TFS-1
               M     : WRITE
               BUS   : ALU
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

04A   SHEAP1   ALU   : A= HBASE  Cn= 1     F= A-D    B NO
                      B= X      D=  IR    Y= F      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

04B   SHEAP2   ALU   : A= TOP-1  Cn= X     F= AV0    B <- F
                      B= TOP-1  D=  X     Y= Aone   Q No
               STACK: TFS <- TFS-1
               M     : WRITE
               BUS   : ALU
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

050   WRITE1   ALU   : A= X      Cn= X     F= DV0    B <- F
                      B= TEMP   D=  IR    Y= None   Q No
               STACK: TFS <- TFS-1
               M     : X
               BUS   : X
               uPU   : CJS  ;  PL = WRT1  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

051   WRITE2   ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

052   WRITEC1  ALU   : A= X      Cn= X     F= DV0    B <- F
                      B= TEMP   D=  IR    Y= None   Q No
               STACK: TFS <- TFS-1
               M     : X
               BUS   : X
               uPU   : CJS  ;  PL = WRTC1  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <-IC+1  ;  IR <- IM(IC)
               IOP   : X
```

```
053   WRITEC2  ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

054   RSTAK1   ALU   : A= TOP-1  Cn= X     F= AV0    B <- F
                       B= TEMP   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = RD1  ;  COND = forced
               IPU   : X
               IOP   : X

056   RSTAKC1  ALU   : A= TOP-1  Cn= X     F= AV0    B <- F
                       B= TEMP   D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = RDC1  ;  COND = forced
               IPU   : X
               IOP   : X
```

The following routines perform the handshaking with the Input/Output processor, to read in or write out one word. They both expect the device code to be in the TEMP register.

The routine to perform output to the device whose device code is TEMP.

```
058   WRT1     ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = WRT1  ;  COND = IORDY*
               IPU   : X
               IOP   : X
                       Wait for the IOP to be "ready"
```

```
059   WRT2      ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                        B= TEMP   D=  X     Y= A      Q No
                STACK: X
                M     : X
                BUS   : ALU
                uPU   : CJP  ;  PL = WRT2  ;  COND = IORDY
                IPU   : X
                IOP   : DEVSEL
                        Put the device code on the bus and raise
                        the handshake signal DEVSEL.  Wait for
                        IOP to acknowledge by going "not ready".

05A   WRT3      ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = WRT3  ;  COND = IORDY*
                IPU   : X
                IOP   : X
                        Remove the device code from the bus.
                        Wait for the IOP to be "ready" again.

05B   WRT4      ALU   : A= TOP    Cn= X     F= AV0    B <- F
                        B= TOP    D=  X     Y= A      Q No
                STACK: X
                M     : X
                BUS   : ALU
                uPU   : CJP  ;  PL = WRT4  ;  COND = IORDY
                IPU   : X
                IOP   : IODATA  ;  WRITE
                        Put data on bus.  Raise I/O signals.
                        Wait for acknowledge by "not ready".

05C   WRT5      ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : CRTN  ;  COND = forced
                IPU   : X
                IOP   : X
                        Do not need to wait for IOP to release
                        the bus on a write operation.

05D   WRTC1     ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = WRT1  ;  COND = IORDY*
                IPU   : X
                IOP   : X
                        Wait for the IOP to be ready.
```

```
05E   WRTC2    ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                      B= TEMP   D=  X     Y= A      Q No
               STACK: X
               M     : X
               BUS   : ALU
               uPU   : CJP  ;  PL = WRTC2  ;  COND = IORDY
               IPU   : X
               IOP   : DEVSEL
                       Put the device code on the bus.

05F   WRTC3    ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = WRTC3 ;  COND = IORDY*
               IPU   : X
               IOP   : X
                       Wait for IOP to respond by going "ready"

060   WRTC4    ALU   : A= TOP    Cn= X     F= AV0    B <- F
                       B= TOP    D=  X     Y= A      Q No
               STACK: X
               M     : X
               BUS   : ALU
               uPU   : CJP  ;  PL = WRT4  ;  COND = IORDY
               IPU   : X
               IOP   : IODATA  ;  WRITE  ;  CHAR
                       Put the data on the buss and wait for
                       IOP to signal its acceptance by going
                       "not ready".

061   WRTC5    ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CRTN ;  COND = forced
               IPU   : X
               IOP   : X

062   RD1      ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = RD1  ;  COND = IORDY*
               IPU   : X
               IOP   : X
                       Wait for IOP to be "ready".
```

```
063   RD2     ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                      B= TEMP   D=  X     Y= A      Q No
              STACK: X
              M     : X
              BUS   : ALU
              uPU   : CJP  ;  PL = RD2  ;  COND = IORDY
              IPU   : X
              IOP   : DEVSEL
                      Put the device code on the bus.  Wait
                      for the IOP to acknowledge by going
                      "not ready".

064   RD3     ALU   : X
              STACK: X
              M     : X
              BUS   : X
              uPU   : CJP  ;  PL = RD3  ;  COND = IORDY*
              IPU   : X
              IOP   : X
                      Remove device code.  Wait for IOP to
                      go "ready" again.

065   RD4     ALU   : X
              STACK: X
              M     : X
              BUS   : IOP
              uPU   : CJP  ;  PL = RD4  ;  COND = IORDY
              IPU   : X
              IOP   : IODATA  ;  READ
                      Give IOP the system bus.  Wait for
                      "not ready" to indicate valid data.

066   RD5     ALU   : A= X      Cn= X     F= DV0    B <- F
                      B= TOP-1  D=  BUS   Y= None   Q No
              STACK: X
              M     : X
              BUS   : IOP
              uPU   : CONT
              IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : IODATA  ;  READ
                      Read in the data.

067   RD6     ALU   : X
              STACK: X
              M     : X
              BUS   : IOP
              uPU   : CJP  ;  PL = RD6  ;  COND = IORDY*
              IPU   : X
              IOP   : X
                      Wait for IOP to release the bus.
```

```
068   RD7       ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X

069   RDC1      ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = RD1  ;  COND = IORDY*
                IPU   : X
                IOP   : X
                        Wait for IOP to be "ready".

06A   RDC2      ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                        B= TEMP   D=  X     Y= A      Q No
                STACK: X
                M     : X
                BUS   : ALU
                uPU   : CJP  ;  PL = RD2  ;  COND = IORDY
                IPU   : X
                IOP   : DEVSEL
                        Put the device code on the bus.  Wait
                        for the IOP to acknowledge by going
                        "not ready".

06B   RDC3      ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = RD3  ;  COND = IORDY*
                IPU   : X
                IOP   : X
                        Remove device code.  Wait for IOP to
                        go "ready" again.

06C   RDC4      ALU   : X
                STACK: X
                M     : X
                BUS   : IOP
                uPU   : CJP  ;  PL = RD4  ;  COND = IORDY
                IPU   : X
                IOP   : IODATA  ;  READ  ;  CHAR
                        Give IOP the system bus.  Wait for
                        "not ready" to indicate valid data.
```

```
06D   RDC5     ALU   : A= X       Cn= X      F= DV0     B <- F
                       B= TOP-1   D=  BUS    Y= None    Q No
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CONT
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : IODATA  ;  READ  ;  CHAR
                       Read in the data.

06E   RDC6     ALU   : X
               STACK: X
               M     : X
               BUS   : IOP
               uPU   : CJP  ;  PL = RD6  ;  COND = IORDY*
               IPU   : X
               IOP   : X
                       Wait for IOP to release the bus.

06F   RDC7     ALU   : X
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
```

The following routines all require at least two operands in the fast stack.

```
080   ADD1     ALU   : A= TOP-1  Cn= 0      F= A+B     B <- F
                       B= TOP-2  D=  X      Y= None    Q No
               STACK: TFS <- TFS-1
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Pop and add the top two values.  Push the
                       sum.

082   SUB1     ALU   : A= TOP-1  Cn= 1      F= B-A     B <- F
                       B= TOP-2  D=  X      Y= None    Q No
               STACK: TFS <- TFS-1
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Pop and subtract the top value from the
                       second value.  Push the difference.
```

```
084   MULT1   ALU   : A= TOP-1  Cn= 0     F= AV0    B NO
                      B= X      D=  X     Y= None   Q <- F
              STACK: X
              M     : X
              BUS   : X
              uPU   : LDCT  ;  PL = $1F
              IPU   : X
              IOP   : X
                      Copy the multiplier into a temporary.
                      Load the counter for the number of
                      iterations.

085   MULT2   ALU   : A= X      Cn= X     F= B&0    B <- F
                      B= TEMP   D=  X     Y= None   Q No
              STACK: X
              M     : X
              BUS   : X
              uPU   : CJP  ;  PL = MULT3  ;  COND = forced
              IPU   : X
              IOP   : X
                      Clear the temporary that receives the
                      most significant word of the product.
                      Continued at location 09A.

086   DIV1    ALU   : A= X      Cn= X     F= Q&0    B NO
                      B= X      D=  X     Y= None   Q <- F
              STACK: X
              M     : X
              BUS   : X
              uPU   : LDCT  ;  PL = DIV 3A
              IPU   : X
              IOP   : X
                      Clear quotient.  Load a jump address into
                      the register.

087   DIV2    ALU   : A= TOP-1  Cn= X     F= AV0    B NO
                      B= X      D=  X     Y= None   Q No
              STACK: X
              M     : X
              BUS   : X
              uPU   : JRP  ;  PL = DIV3B  ;  COND = negative
              IPU   : X
              IOP   : X
                      Test the sign of the divisor.
                      Jump to the address in R if positive,
                      and the address in PL if negative.
                      Continued at location 09D.
```

```
088   EQU1      ALU   : A= TOP-1  Cn= 1     F= B-A    B NO
                      B= TOP-2  D=  X     Y= None   Q No
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = LOADT1  ;  COND = equal
                IPU   : X
                IOP   : X
                        Compare the top two stack elements.  Load
                        a TRUE if the values are equal.
                        Otherwise load a FALSE.

089   LOADF1    ALU   : A= FALSE  Cn= X     F= AV0    B <- F
                      B= TOP-2  D=  X     Y= None   Q No
                STACK: TFS <- TFS-1
                M     : X
                BUS   : X
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X
                        Load a FALSE logic value onto the stack.

08A   GREAT1    ALU   : A= TOP-1  Cn= 1     F= B-A    B NO
                      B= TOP-2  D=  X     Y= None   Q No
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = LOADF1  ;  COND = not equal
                IPU   : X
                IOP   : X
                        Compare the top two stack elements.  Load
                        a logic TRUE value if the second
                        element is larger than the first.
                        Otherwise load a FALSE.

08B   LOADT1    ALU   : A= TRUE   Cn= X     F= AV0    B <- F
                      B= TOP-2  D=  X     Y= None   Q No
                STACK: TFS <- TFS-1
                M     : X
                BUS   : X
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X
                        Load a TRUE value onto the stack.

08C   LESS1     ALU   : A= TOP-1  Cn= 1     F= B-A    B NO
                      B= TOP-2  D=  X     Y= None   Q No
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = LOADT1  ;  COND = positive
                IPU   : X
                IOP   : X
                        Compare the top two stack elements.  Load
```

```
                    a TRUE value onto the stack if the second
                    value is less than the top value.
                    Otherwise load a FALSE value.

08D   LESS2   ALU   : A= FALSE  Cn= X     F= AV0    B <- F
                      B= TOP-2  D=  X     Y= None   Q No
              STACK: TFS <- TFS-1
              M     : X
              BUS   : X
              uPU   : JMAP
              IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : X
                      Load the FALSE value.

08E   AND1    ALU   : A= TOP-1  Cn= X     F= B&A    B <- F
                      B= TOP-2  D=  X     Y= None   Q No
              STACK: TFS <- TFS-1
              M     : X
              BUS   : X
              uPU   : JMAP
              IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : X
                      AND the top two stack elements.

090   OR1     ALU   : A= TOP-1  Cn= X     F= AVB    B <- F
                      B= TOP-2  D=  X     Y= None   Q No
              STACK: TFS <- TFS-1
              M     : X
              BUS   : X
              uPU   : JMAP
              IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : X
                      OR the top two stack elements.

092   REP1    ALU   : A= TOP-2  Cn= X     F= AV0    B <- F
                      B= TOP-2  D=  X     Y= Aone   Q No
              STACK: TFS <- TFS-1
              M     : LOAD MAR
              BUS   : ALU
              uPU   : CONT
              IPU   : X
              IOP   : X
                      Load the address one down from the top
                      of the stack onto the MAR.

093   REP2    ALU   : A= X      Cn= X     F= DV0    B <- F
                      B= TOP-1  D=  BUS   Y= None   Q No
              STACK: X
              M     : READ
              BUS   : M
              uPU   : JMAP
              IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : X
                      Fetch the value stored at that address.
```

```
                    Save it at the location that the address
                    came from.  Remove the element that was
                    at the top.

094   STOR1   ALU   : A= TOP-2  Cn= X     F= AV0    B <- F
                      B= TOP-2  D=  X     Y= A      Q No
              STACK: TFS <- TFS-1
              M     : LOAD MAR
              BUS   : ALU
              uPU   : CONT
              IPU   : X
              IOP   : X
                    Load the address from the top of the
                    stack into the MAR.

095   STOR2   ALU   : A= TOP    Cn= X     F= AV0    B <- F
                      B= TOP    D=  X     Y= A      Q No
              STACK: TFS <- TFS-1
              M     : WRITE
              BUS   : ALU
              uPU   : JMAP
              IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : X
                    Then write out the data at the new top of
                    the stack.

096   WSTAK1  ALU   : A= TOP-2  Cn= X     F= AV0    B <- F
                      B= TEMP   D=  X     Y= None   Q No
              STACK: TFS <- TFS-1
              M     : X
              BUS   : X
              uPU   : CJS  ;  PL = WRT1  ;  COND = forced
              IPU   : X
              IOP   : X
                    Get the device code out from one down
                    from the top of the stack.  Put it where
                    the write routine can find it.

097   WSTAK2  ALU   : A= X
              STACK: TFS <- TFS-1
              M     : X
              BUS   : X
              uPU   : JMAP
              IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP   : X
                    Do the bookkeeping.
```

```
098   WSTAKC1   ALU   : A= TOP-2  Cn= X     F= AV0    B <- F
                      B= TEMP   D=  X     Y= None   Q No
                STACK: TFS <- TFS-1
                M     : X
                BUS   : X
                uPU   : CJS  ;  PL = WRT1  ;  COND = forced
                IPU   : X
                IOP   : X
                        Same as WSTAK1 except calls WRTC1.

099   WSTAKC2   ALU   : X
                STACK: TFS <- TFS-1
                M     : X
                BUS   : X
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X
```

The following routines are continuations of routines that require two operands.

```
09A   MULT3     ALU   : A= TOP-2  Cn= 0     F= A+B    B <- F/2
                        B= TEMP   D=  X     Y= None   Q <- Q/2
                        MULT
                STACK: X
                M     : X
                BUS   : X
                uPU   : RPCT  ;  PL = MULT3
                IPU   : X
                IOP   : X
                        Repeat this step for each bit in the
                        multiplicand.  Conditional shift and
                        add.

09B   MULT4     ALU   : A= TOP-2  Cn= 1     F= B-A    B <- F/2
                        B= TEMP   D=  X     Y= None   Q <- Q/2
                        MULT
                STACK: X
                M     : X
                BUS   : X
                uPU   : CONT
                IPU   : X
                IOP   : X
                        Conditional shift and subtract for the
                        MSB.  This is what makes it two's
                        complement.
```

```
09C   MULT5    ALU   : A= X      Cn= 0     F= QV0    B <- F
                      B= TOP-2   D=  X     Y= None   Q No
               STACK: TFS <- TFS-1
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Use the LSW of the result.  Move it into
                       the top of the stack.

09D   DIV3A    ALU   : A= TOP-2   Cn= X     F= AV0    B NO
                      B= X       D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV4C ;  COND = negative
               IPU   : X
               IOP   : X
                       Test the sign of the dividend.

09E   DIV4A    ALU   : A= X       Cn= 1     F= Q+0    B NO
                      B= X       D=  X     Y= NONE   Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Count the number of subtractions.

09F   DIV5A    ALU   : A= TOP-1   Cn= 1     F= B-A    B <- F
                      B= TOP-2   D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV4A ;  COND = not negative
               IPU   : X
               IOP   : X
                       Subtract until the result is negative.

0A0   DIV6A    ALU   : A= X       Cn= 0     F= Q-0    B NO
                      B= X       D=  X     Y= NONE   Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Add 1 back in 'cause went one too far.
```

```
0A1   DIV7A    ALU   : A= TOP-1  Cn= 0     F= B+A    B <- F
                       B= TOP-2  D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV8  ;  COND = forced
               IPU   : X
               IOP   : X
                       Move the quotient onto the stack.

0A2   DIV4C    ALU   : A= X      Cn= 0     F= Q-0    B NO
                       B= X      D=  X     Y= NONE   Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Count the number of subtracts (backwards)

0A3   DIV5C    ALU   : A= TOP-1  Cn= 0     F= B+A    B <- F
                       B= TOP-2  D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV4C ;  COND = not positive
               IPU   : X
               IOP   : X
                       Subtract the divisor (really add because
                       of the signs.)

0A4   DIV6C    ALU   : A= X      Cn= 1     F= Q+0    B NO
                       B= X      D=  X     Y= NONE   Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Change quotient because went one too far.

0A5   DIV7C    ALU   : A= TOP-1  Cn= 0     F= B-A    B <- F
                       B= TOP-2  D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV8  ;  COND = forced
               IPU   : X
               IOP   : X
                       Add it back in to get proper result.
```

```
0A6   DIV3B    ALU   : A= TOP-1  Cn= 0     F= AV0    B NO
                       B= X      D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV4D ;  COND = negative
               IPU   : X
               IOP   : X
                       Check the sign of the dividend.

0A7   DIV4B    ALU   : A= X      Cn= 0     F= Q-0    B NO
                       B= X      D=  X     Y= NONE   Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X

0A8   DIV5B    ALU   : A= TOP-1  Cn= 0     F= B+A    B <- F
                       B= TOP-2  D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV4B ;  COND = not negative
               IPU   : X
               IOP   : X

0A9   DIV6B    ALU   : A= X      Cn= 1     F= Q+0    B NO
                       B= X      D=  X     Y= NONE   Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X

0AA   DIV7B    ALU   : A= TOP-1  Cn= 0     F= B-A    B <- F
                       B= TOP-2  D=  X     Y= NONE   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV8  ;  COND = forced
               IPU   : X
               IOP   : X
```

```
0AB   DIV4D    ALU   : A= X       Cn= 1      F= Q+0     B NO
                       B= X       D=  X      Y= NONE    Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X

0AC   DIV5D    ALU   : A= TOP-1   Cn= 0      F= B-A     B <- F
                       B= TOP-2   D=  X      Y= NONE    Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV4D ;  COND = positive
               IPU   : X
               IOP   : X

0AD   DIV6D    ALU   : A= X       Cn= 0      F= Q-0     B NO
                       B= X       D=  X      Y= NONE    Q <- F
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X

0AE   DIV7D    ALU   : A= TOP-1   Cn= 0      F= B+A     B <- F
                       B= TOP-2   D=  X      Y= NONE    Q NO
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DIV8  ;  COND = forced
               IPU   : X
               IOP   : X

0AF   DIV8     ALU   : A= X       Cn= 0      F= QV0     B <- F
                       B= TOP-1   D=  X      Y= NONE    Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <-IM(IC)
               IOP   : X
```

The following routines all require that the fast stack have at least one empty location before they start.

```
100   LCONST1   ALU   : A= X       Cn= X      F= DV0     B <- F
                        B= TOP     D=  IR     Y= None    Q No
                STACK: TFS <- TFS+1
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = DLY1  ;  COND = forced
                IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X

101   LADD3     ALU   : A= TEMP    Cn= 0      F= D+A     B <- F
                        B= TOP     D=  IR     Y= None    Q No
                STACK: TFS <- TFS+1
                M     : READ
                BUS   : M
                uPU   : JMAP
                IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X
                        Calculate the address as (LL(11)) + j

102   LADD1     ALU   : A= DISP    Cn= 0      F= D+A     B NO
                        B= X       D=  IR     Y= F       Q No
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CONT
                IPU   : IR = MASKED(IR2) ; IC <- IC+1 ;
                        IR <- IM(IC)
                IOP   : X
                        Fetch the contents of the display
                        register.

103   LADD2     ALU   : A= X       Cn= X      F= DV0     B <- F
                        B= TEMP    D=  IR     Y= None    Q No
                STACK: X
                M     : X
                BUS   : X
                uPU   : CJP  ;  PL = LADD3  ;  COND = forced
                IPU   : IR = IR2
                IOP   : X

104   LDATA1    ALU   : A= DISP    Cn= 0      F= D+A     B NO
                        B= X       D=  IR     Y= F       Q No
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CJS  ;  PL = ADDEV1  ;  COND = forced
                IPU   :  IR = MASKED(IR2)  ;  IC <- IC+1  ;
                        IR <- IM(IC)
                IOP   : X
                        Calculate the address of the display
                        register and load it into the MAR.
                        Call ADDEV to evaluate the variable
                        address.
```

```
105   LDATA2   ALU   : A= X       Cn= X      F= DV0     B <- F
                       B= TOP     D=  BUS    Y= None    Q No
               STACK: TFS <- TFS+1
               M     : READ
               BUS   : M
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Fetch the value from the address.  Store
                       it onto the top of the stack.

106   LHEAP1   ALU   : A= HBASE   Cn= 1      F= A-D     B NO
                       B= X       D=  IR     Y= F       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Calculate the address in the heap.  Load
                       it into the MAR.

107   LHEAP2   ALU   : A= X       Cn= X      F= DV0     B <- F
                       B= TOP     D=  BUS    Y= None    Q No
               STACK: TFS <- TFS+1
               M     : READ
               BUS   : M
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Fetch the value from that address and
                       load it onto the stack.

108   LHADD1   ALU   : A= HBASE   Cn= 1      F= A-D     B <- F
                       B= TOP     D=  IR     Y= None    Q No
               STACK: TFS <- TFS+1
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DLY1  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Calculate the address of a variable
                       in the heap.  Load the address onto the
                       top of the stack.

10A   READ1    ALU   : A= X       Cn= X      F= DV0     B <- F
                       B= TEMP    D=  IR     Y= None    Q No
               STACK: TFS <- TFS+1
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = RD1  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Read the device code from the instruction
```

```
                    stream.  Put it where the read routine
                    can find it.  Do the stack maintenance.

10C   READC1   ALU  : A= X      Cn= X     F= DV0    B <- F
                     B= TEMP   D=  IR    Y= None   Q No
              STACK: TFS <- TFS+1
              M    : X
              BUS  : X
              uPU  : CJP  ;  PL = RDC1  ;  COND = forced
              IPU  :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP  : X
```

The following routines all require that the fast stack be empty before they begin execution.

```
1C0   LLAB1     ALU  : A= DISP   Cn= 1     F= A+B    B NO
                     B= LLTOP  D=  X     Y= F      Q No
              STACK: X
              M    : LOAD MAR
              BUS  : ALU
              uPU  : CONT
              IPU  :  IR = X  ;  IC <- IC+1  ;  IR <- IM(IC)
              IOP  : X
                     Get the current display register.

1C1   LLAB2     ALU  : A= X      Cn= X     F= DV0    B <- F
                     B= TEMP   D=  BUS   Y= None   Q No
              STACK: X
              M    : READ
              BUS  : M
              uPU  : CJP  ;  PL = LLAB3  ;  COND = forced
              IPU  : X
              IOP  : X

1C2   INCS1     ALU  : A= TMS    Cn= 0     F= A-0    B <- F
                     B= TOP    D=  X     Y= F      Q No
              STACK: TFS <- TFS+1
              M    : LOAD MAR
              BUS  : ALU
              uPU  : CONT
              IPU  : X
              IOP  : X
                     Need to save the old top of the stack
                     at the new top of the stack.  Need the
                     actual value of the top of the stack, so
                     use TMS-1.  A trick here.  The Fast stack
                     is loaded with the value of the old top
                     of the stack.  Whatever happens next will
                     write out the value.
```

```
1C3   INCS2    ALU   : A= TMS    Cn= 0     F= A+D    B <- F
                       B= TMS    D=  BUS   Y= None   Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Fetch the value to add to the stack and
                       add it to TMS.

1C4   ISTAK1   ALU   : A= TMS    Cn= 0     F= A+D    B <- F
                       B= TMS    D=  IR    Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DLY1  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Add the next instruction word to the
                       top of stack pointer.

1C6   DECS1    ALU   : A= TMS    Cn= 0     F= A-0    B NO
                       B= X      D=  X     Y= F      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Same as INCS except subtract.

1C7   DECS2    ALU   : A= TMS    Cn= 1     F= A-D    B <- F
                       B= TOP    D=  BUS   Y= None   Q No
               STACK: TFS <- TFS+1
               M     : READ
               BUS   : M
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

1C8   DSTAK1   ALU   : A= TMS    Cn= 1     F= A-D    B <- F
                       B= TMS    D=  IR    Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = DLY1  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
```

```
1CA   BLKEN1   ALU   : A= TMS     Cn= 1      F= A+0     B <- F
                     B= TMS     D=  X      Y= A       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Save what will be the stack pointer when
                       this instruction finishes.

1CB   BLKEN2   ALU   : A= TMS     Cn= 1      F= A+D     B <- F
                     B= TEMP    D=  IR     Y= F       Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CJP  ;  PL = BLKEN3  ;  COND = forced
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

1CC   BLKEX1   ALU   : A= DISP    Cn= 0      F= A+B     B NO
                     B= LLTOP   D=  X      Y= F       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Restore the stack pointer to what it was
                       before the block was entered.  (i.e. two
                       below where the current display register
                       points)

1CD   BLKEX2   ALU   : A= TWO     Cn= 1      F= D-A     B <- F
                     B= TMS     D=  BUS    Y= None    Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CJP  ;  PL = BLKEX3  ;  COND = forced
               IPU   : X
               IOP   : X

1CE   PROCEN1  ALU   : A= X       Cn= X      F= DV0     B <- F
                     B= LLTOP   D=  IR     Y= None    Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   :  IR = MASKED(IR2)  ;  IC <- IC+1  ;
                       IR <- IM(IC)
               IOP   : X
                       Set the lexical level to that in
                       which the procedure was declared.
```

```
1CF   LABEN3   ALU   : A= TWO    Cn= 1     F= D-A    B <- F
                       B= TMS    D=  BUS   Y= None   Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : JMAP
               IPU   :  IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X

1D0   LABEN1   ALU   : A= X      Cn= X     F= DV0    B <- F
                       B= LLTOP  D=  IR    Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : IR = MASKED(IR2)
               IOP   : X
                       Set the current lexical level to that of
                       the destination.

1D1   LABEN2   ALU   : A= DISP   Cn= 0     F= A+B    B NO
                       B= LLTOP  D=  X     Y= None   Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CJP  ;  PL = LABEN3  ;  COND = forced
               IPU   : X
               IOP   : X
                       Set the top of stack pointer from the
                       value saved by the destinations block
                       entry.

1D2   JIND1    ALU   : A= DISP   Cn= 0     F= A+D    B NO
                       B= X      D=  IR    Y= F      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CJS  ;  PL = ADDEV1  ;  COND = forced
               IPU   :  IR = MASKED (IR2)  ;  IC <- IC+1  ;
                       IR <- IM(IC)
               IOP   : X
                       Let the subroutine evaluate the address
                       of the variable which points to the
                       transfer point.

1D3   JIND2    ALU   : A= X      Cn= X     F= DV0    B NO
                       B= X      D=  BUS   Y= None   Q <- F
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CJP  ;  PL = SCOPE1  ;  COND = forced
               IPU   : X
               IOP   : X
                       Put the address of the transfer point
```

```
                    into the Q register where the routine
                    which sets up the scope of the
                    destination by tracing the display
                    chain expects it.
1D4   CALL1   ALU  : A= DISP   Cn= 0     F= A+B    B NO
                     B= LLTOP  D=  X     Y= F      Q No
              STACK: X
              M    : LOAD MAR
              BUS  : ALU
              uPU  : CONT
              IPU  : X
              IOP  : X
                    Store a transfer point on the stack.

1D5   CALL2   ALU  : A= X      Cn= X     F= DV0    B <- F
                     B= TEMP   D=  BUS   Y= None   Q No
              STACK: X
              M    : READ
              BUS  : M
              uPU  : CJP  ;  PL = CALL3  ;  COND = forced
              IPU  : X
              IOP  : X

1D6   CALIN1  ALU  : A= DISP   Cn= 0     F= A+B    B NO
                     B= LLTOP  D=  X     Y= F      Q No
              STACK: X
              M    : LOAD MAR
              BUS  : ALU
              uPU  : CONT
              IPU  : X
              IOP  : X
                    Save a transfer point.

1D7   CALIN2  ALU  : A= X      Cn= X     F= DV0    B <- F
                     B= TEMP   D=  BUS   Y= None   Q No
              STACK: X
              M    : READ
              BUS  : M
              uPU  : CJP  ;  PL = CALIN3  ;  COND = forced
              IPU  : X
              IOP  : X

1D8   RET1    ALU  : A= TMS    Cn= X     F= AV0    B NO
                     B= X      D=  X     Y= None   Q <- F
              STACK: X
              M    : X
              BUS  : X
              uPU  : CONT
              IPU  : X
              IOP  : X
                    Copy the top of stack pointer into Q.
                    The Scope setting routine requires the
                    address of the transfer point in Q.
```

```
1D9   RET2     ALU   : A= TWO    Cn= 0     F= B-A    B <- F
                       B= TMS    D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = RTN3  ;  COND = forced
               IPU   : X
               IOP   : X
                       Remove the transfer point from the stack
                       by decrementing the top of stack pointer
                       by two.

1DA   RET3     ALU   : A= X      Cn= 1     F= B-D    B <- F
                       B= TMS    D=  IR    Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CJP  ;  PL = SCOPE1  ;  COND = forced
               IPU   : IR = IR1
               IOP   : X

1DB   BLKEX3   ALU   : A= LLTOP  Cn= 0     F= A-0    B <- F
                       B= LLTOP  D=  BUS   Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Decrement the current lexical level.

1DC   LLAB3    ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                       B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1DD   LLAB4    ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                       B= TEMP   D=  X     Y= A      Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1DE   LLAB5    ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                       B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
```

```
                uPU   : CONT
                IPU   : X
                IOP   : X

1DF   LLAB6     ALU   : A= LLTOP  Cn= X     F= AV0    B <- F
                        B= LLTOP  D=  X     Y= A      Q No
                STACK: X
                M     : WRITE
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X

1E0   LLAB7     ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                        B= TMS    D=  X     Y= A      Q No
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X

1E1   LLAB8     ALU   : X
                STACK: X
                M     : WRITE
                BUS   : IR
                uPU   : CONT
                IPU   : IR = IR2
                IOP   : X

1E2   LLAB9     ALU   : X
                STACK: X
                M     : X
                BUS   : X
                uPU   : JMAP
                IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
                IOP   : X

1E3   BLKEN3    ALU   : LLTOP     Cn= 0     F= A+B    B NO
                        B= DISP   D=  X     Y= F      Q No
                STACK: X
                M     : LOAD MAR
                BUS   : ALU
                uPU   : CONT
                IPU   : X
                IOP   : X
                        Get the current display register.
```

```
1E4   BLKEN4   ALU   : A= X       Cn= X     F= DV0     B <- F
                      B= TEMP2    D=  BUS   Y= None    Q No
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CONT
               IPU   : X
               IOP   : X
                      Save it for a little bit.

1E5   BLKEN5   ALU   : A= TMS     Cn= 1     F= A+0     B <- F
                      B= TMS     D=  X     Y= A       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1E6   BLKEN6   ALU   : A= TEMP2   Cn= X     F= AV0     B <- F
                      B= TEMP2   D=  X     Y= A       Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                      Put it on the stack.
1E7   BLKEN7   ALU   : A= X       Cn= 1     F= B+0     B <- F
                      B= LLTOP   D=  X     Y= None    Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : CONT
               IPU   : X
               IOP   : X
                      Increment the current lexical level.

1E8   BLKEN8   ALU   : A= DISP    Cn= 0     F= A+B     B NO
                      B= LLTOP   D=  X     Y= F       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
```

```
1E9   BLKEN9   ALU   : A= TMS    Cn= X     F= AV0    B <- F
                     B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Load the new display register with the
                       stack top pointer.

1EA   BLKEN10  ALU   : A= TEMP2  Cn= X     F= AV0    B <- F
                     B= TMS    D=  X     Y= None   Q No
               STACK: X
               M     : X
               BUS   : X
               uPU   : JMAP
               IPU   : IR = IR1  ;  IC <- IC+1  ;  IR <- IM(IC)
               IOP   : X
                       Load the top of stack pointer with the
                       working pointer value calculated earlier.
                       This allocates the variables for the
                       block.

1EB   CALL3    ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                     B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1EC   CALL4    ALU   : A= TEMP   Cn= X     F= AV0    B <- F
                     B= TEMP   D=  X     Y= A      Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Save the current display register on
                       the stack.

1ED   CALL5    ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                     B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
```

```
1EE   CALL6    ALU   : A= LLTOP  Cn= X     F= AV0    B <- F
                       B= LLTOP  D=  X     Y= A      Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
                       Save the lexical level of the calling
                       block.

1EF   CALL7    ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                       B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1F0   CALL8    ALU   : X
               STACK: X
               M     : WRITE
               BUS   : IC
               uPU   : CONT
               IPU   : SAVEIC
               IOP   : X
                       Save the return address.

1F1   CALL9    ALU   : X
               STACK: X
               M     : X
               BUS   : IR
               uPU   : CJP  ;  PL = DLY1  ;  COND = forced
               IPU   : IR = IR1  ;  JUMP  ;  forced
               IOP   : X
                       Load the address of the procedure into
                       the Instruction Counter, then give the
                       mapping logic time to decode it
                       destination instruction.

1F2   CALIN3   ALU   : A= TMS    Cn= 1     F= A+0    B <- F
                       B= TMS    D=  X     Y= A      Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X
```

```
1F3   CALIN4   ALU   : A= X       Cn= X      F= BV0     B NO
                      B= TEMP    D=  X      Y= F       Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1F4   CALIN5   ALU   : A= TMS     Cn= 1      F= A+0     B <- F
                      B= TMS     D=  X      Y= A       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1F5   CALIN6   ALU   : A= X       Cn= X      F= BV0     B NO
                      B= LLTOP   D=  X      Y= F       Q No
               STACK: X
               M     : WRITE
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1F6   CALIN7   ALU   : A= TMS     Cn= 1      F= A+0     B <- F
                      B= TMS     D=  X      Y= A       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CONT
               IPU   : X
               IOP   : X

1F7   CALIN8   ALU   : X
               STACK: X
               M     : WRITE
               BUS   : IR
               uPU   : CONT
               IPU   : SAVEIC
               IOP   : X

1F8   CALIN9   ALU   : A= DISP    Cn= 0      F= A+D     B NO
                      B= X       D=  IR     Y= F       Q No
               STACK: X
               M     : LOAD MAR
               BUS   : ALU
               uPU   : CJS  ;  PL = ADDEV1  ;  COND = forced
               IPU   :  IR = MASKED(IR2)  ;  IC <- IC+1  ;
                        IR <- IM(IC)
               IOP   : X
```

```
                        Get the address of the variable that
                        contains the address of the transfer
                        point.

1FA   CALIN10  ALU   : A= X      Cn= X     F= DV0    B NO
                       B= X      D=  BUS   Y= None   Q <- F
               STACK: X
               M     : READ
               BUS   : M
               uPU   : CJS  ;  PL = SCOPE1  ;  COND = forced
               IPU   : X
               IOP   : X
                        Jump to the subroutine that establishes
                        the scope of the destination given the
                        address of a transfer point in the Q
                        register.
```

XIII. CONCLUSIONS

This paper presented the design of a computer that could be built. There are some features that need to be added to the design before the machine could be considered complete. Interrupt capabilities and the ability to manipulate characters as single byte values are the two features that are most notably absent.

There is room in each data word for four character values. If the characters were packed four to a word, the memory requirements for a character string would be reduced by three-fourths. Implementation of this character packing would require changes to the memory, the fast stack control logic, and the ALU. Each reference to the top stack element would require knowledge of the type of operand. This could be done with additional instructions. e.g. EQUAL CHAR to compare the top two stack bytes for equality. The stack would have to grow or shrink by either four bytes or one byte. The design would have to handle the case of a partial word used for characters being followed by a numeric value. The numeric value would have to be stored such that part of it filled the remaining portion of the partial word, or the hardware would have to detect the "hole" in the stack.

Interrupts should also be added to make this machine complete. One simple way of doing this would be to decode the control signals to the micro-sequencer (the I's). When the code for a JUMP VIA MAP instruction is detected, the code for a

CONDITIONAL JUMP VIA VECTOR could be injected if there were an interrupt waiting. The address that is passed to the micro-sequencer could always be the same address (the micro-code routine would have to figure out the interrupting device), or the interrupting device cause the interrupt service routine's address to be put on the VECT input lines.

Other additions that would make the machine more useful include a micro-code garbage collection routine. The routine could be triggered by an interrupt signalling that the heap had become full.

This design has not been built or simulated, nor has the micro-code given been tested or simulated.

XIV. REFERENCES

(1) Mick and Brick, "Bit-Slice Microprocessor Design", McGraw-Hill, New York (1980)

(2) Greenfield and Wray, "Using Microprocessors and Microcomputers: The 6800 Family", John Wiley and Sons, New York (1981)

(3) Thomas Standish, "Data Structure Techniques", pp. 210-232, Addison-Wesley, Reading, Mass. (1980)

(4) Barrett and Couch, "Compiler Construction: Theory and Practice", Science Research Associates (1979)

(5) Siewiorek, Bell and Newell, "Computer Structures: Principles and Examples", pp 178-179, McGraw-Hill, New York (1982)

THE ARCHITECTURE AND DESIGN OF A HIGH LEVEL LANGUAGE PROCESSOR

by

ROGER BREES

B.S., Kansas State University, 1982

AN ABSTRACT OF A MASTERS'S THESIS

submitted in partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

ABSTRACT

This paper presents the architecture and design of a computer supporting the high level, block structured programming languages. The design is aimed at the reader who has some background in digital design procedures. It includes such features as micro-programmed control, separate instruction and data memories, a built in stack and heap mechanism, full variable scoping, a high level machine instruction set, and bit-slice implementation. These features should make the machine useful as an educational tool.

The Arithmetic and logic unit is built around the AMD2901 4 bit slice. Four of the on-board registers in the 2901 are used as a fast stack. The remaining registers are used as temporaries, constants, and to define the stack in memory.

The Memory allows upto 4 Gwords of 32 bits each. There is no byte or half word addressing.

The instruction processing unit controls the fetching of the macro instructions. The instructions are stored in the Instruction Memory. The instruction fetches are pipelined, so the next instruction is waiting at the inputs to the Instruction Register when the current instruction finishes.

The micro-processing unit controls the fetching of the micro-instructions. The macro-instruction opcode is mapped into a micro-code entry address. If the fast stack contains

too many or too few operands for the desired operation, the mapping logic forces a fix address instead of the mapped opcode. The micro-instructions are stored in the Control Store. They are latched into the Pipeline register which supplies the control signals for the machine.

The stack controller maintains the pointers and counters that define the fast stack in the ALU registers. The memory stack is defined by pointers that are kept in the ALU.

The input/output processor handles the IO and provides the console interface. The console is the master controlling device. It can halt the rest of the system and load a value into any of the three memories, load a value into either of the pipelne registers, load a value into the macro-instruction counter, or force the execution of a single micro-instruction. The IO processor allows the input or output to take place in parallel with the program execution.

Features that make this machine useful, especially in an educational aspect are the architecture based on the stack concept, the support for variable scope at the machine level and the high level machine instruction set.